Patrones de Diseño – Concepto de Framework

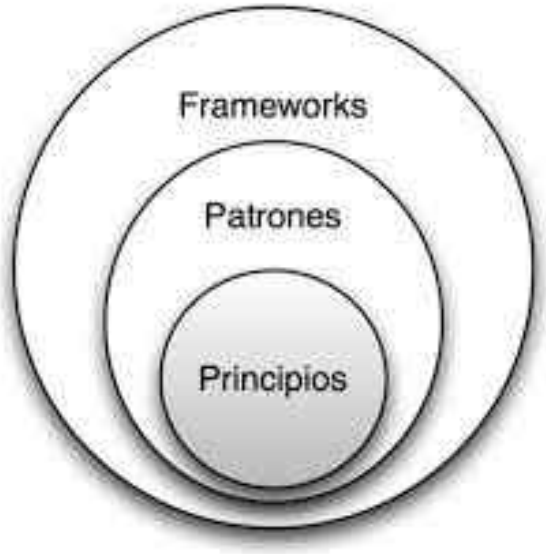
Concepto de Patrón de diseño (Design Pattern).

Ejemplo de Patrones:

- El Patrón Factory
- El Patrón Singleton
- El patrón Observer-Observable. Interfaz Observer. Clase Observable.
- El Patrón Decorator
- El Patrón State
- El Patrón Command

Concepto de Framework.

Patrones de Diseño – Concepto de Framework



Objetivo: la reutilización del software

El diseño O.O. es difícil y el diseño de software orientado a objetos reutilizable lo es aún más.

Los diseñadores expertos no resuelven los problemas desde sus principios; reutilizan soluciones que han funcionado en el pasado.

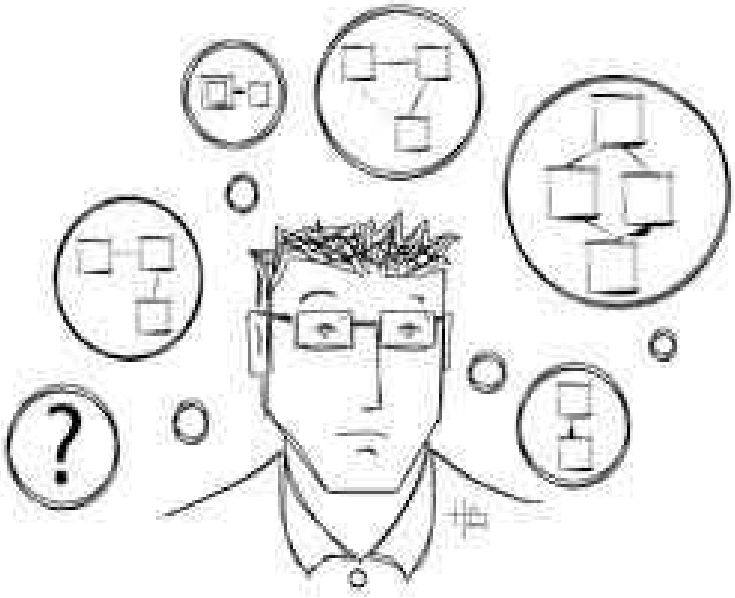
- Se encuentran patrones de clases y objetos de comunicación recurrentes en muchos sistemas orientados a objetos.
- Estos patrones resuelven problemas de diseño específicos y hacen el diseño flexible y reusable.

Patrones de Diseño - definición

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de la solución al problema, de forma que puede utilizarse un millón de veces sin tener que hacer dos veces lo mismo.

Alexander (arquitecto/urbanista)

Patrones de Diseño - definición

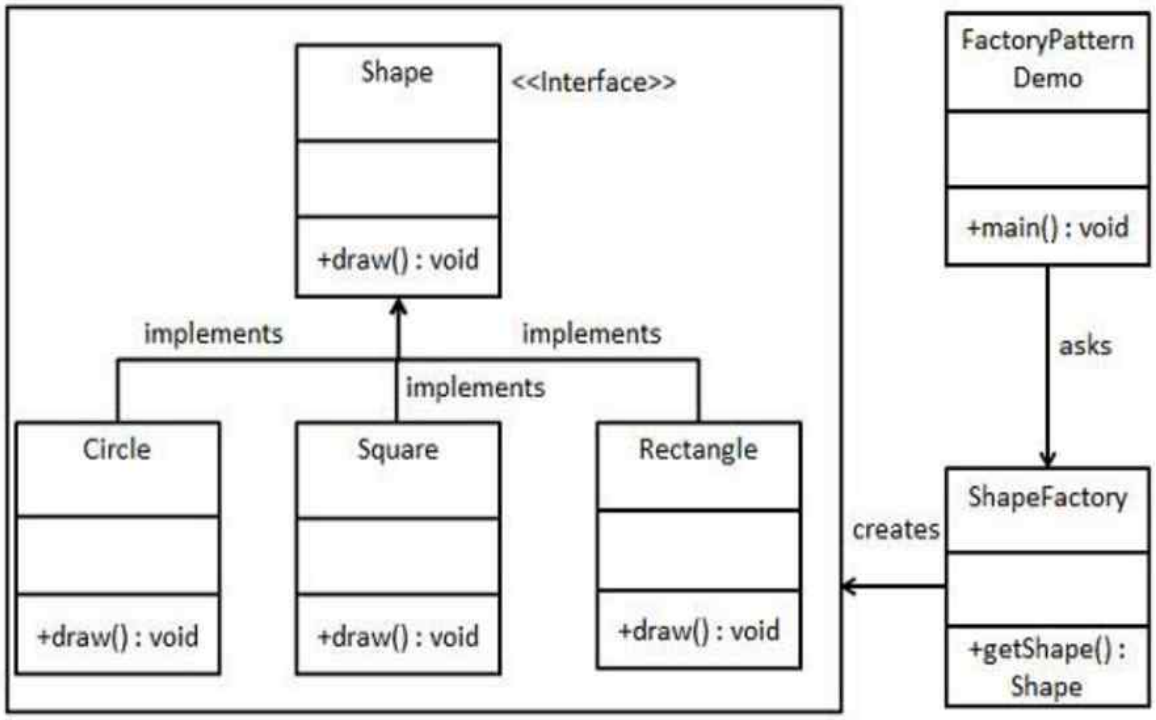


Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular.

Gamma

Un ejemplo rápido:
Patrón Factory

Un ejemplo rápido: el Patrón Factory



Es de tipo *creacional (*1)*

Provee uno de los mejores modos de crear un objeto.

En el patrón Factory, se crea un objeto sin exponer la lógica de la creación al cliente y retorna una referencia al nuevo objeto creado, usando una interfaz común.

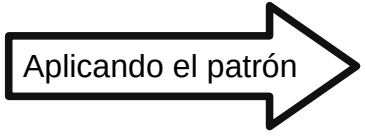
*(*1) los patrones se clasifican según su utilidad*

Un ejemplo rápido: el Patrón Factory

Los patrones creacionales o de creación son aquellos en los que **se delega la instanciación** de un objeto en otro, en lugar de recurrir a un simple new(). La pregunta que nos hacemos es: ¿por qué hacer esto? ¿Qué interés práctico puede existir en crear una clase cuya función sea instanciar otras clases pudiendo dejarle el trabajo a la clase original?

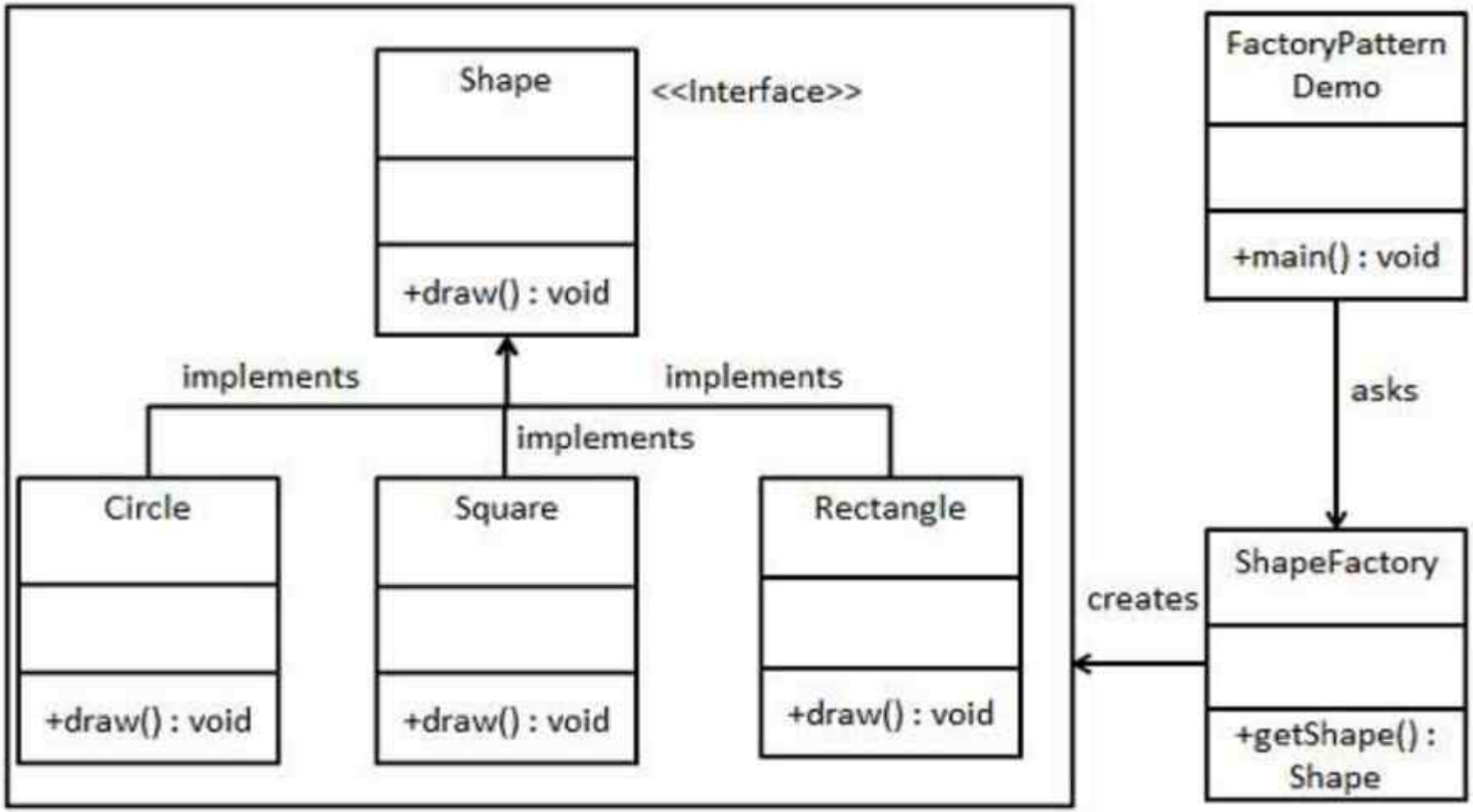
Esta forma de trabajar puede ser útil en algunos escenarios, pero el principal **suele involucrar el no saber qué objeto vamos a instanciar hasta el momento de la ejecución.** Valiéndonos del polimorfismo podremos utilizar una interfaz para alojar una referencia a un objeto que será instanciado por un tercero en lugar de dejar que sea el propio constructor del objeto el que proporcione la instancia.
Por tanto, **nuestro objetivo principal será la encapsulación de la creación de objetos.**

```
Shape forma = new Circle();
```




```
Shape forma = ShapeFactory.getShape("CIRCLE");
```

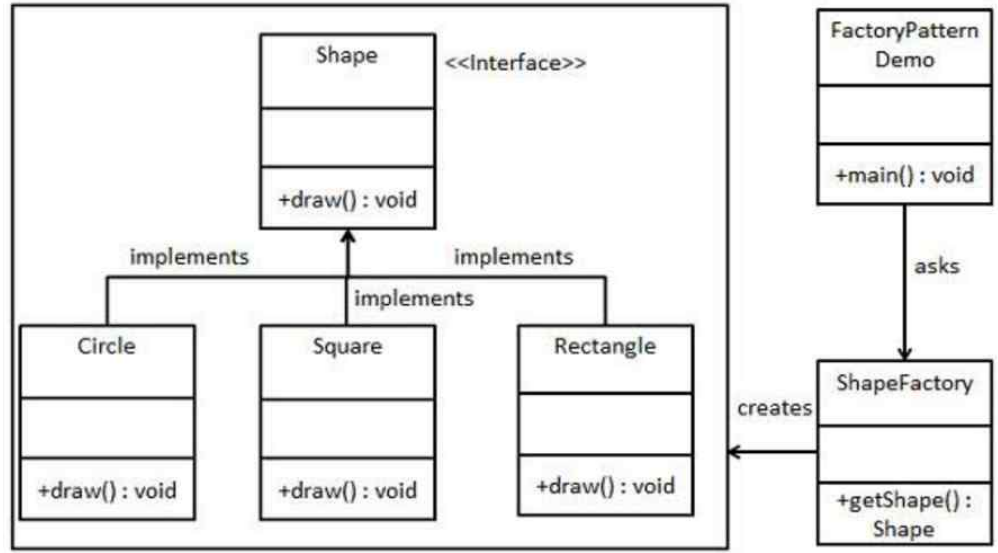
Un ejemplo rápido: el Patrón Factory



Un ejemplo rápido: el Patrón Factory

```java
public interface Shape
{
   void draw();
}
```

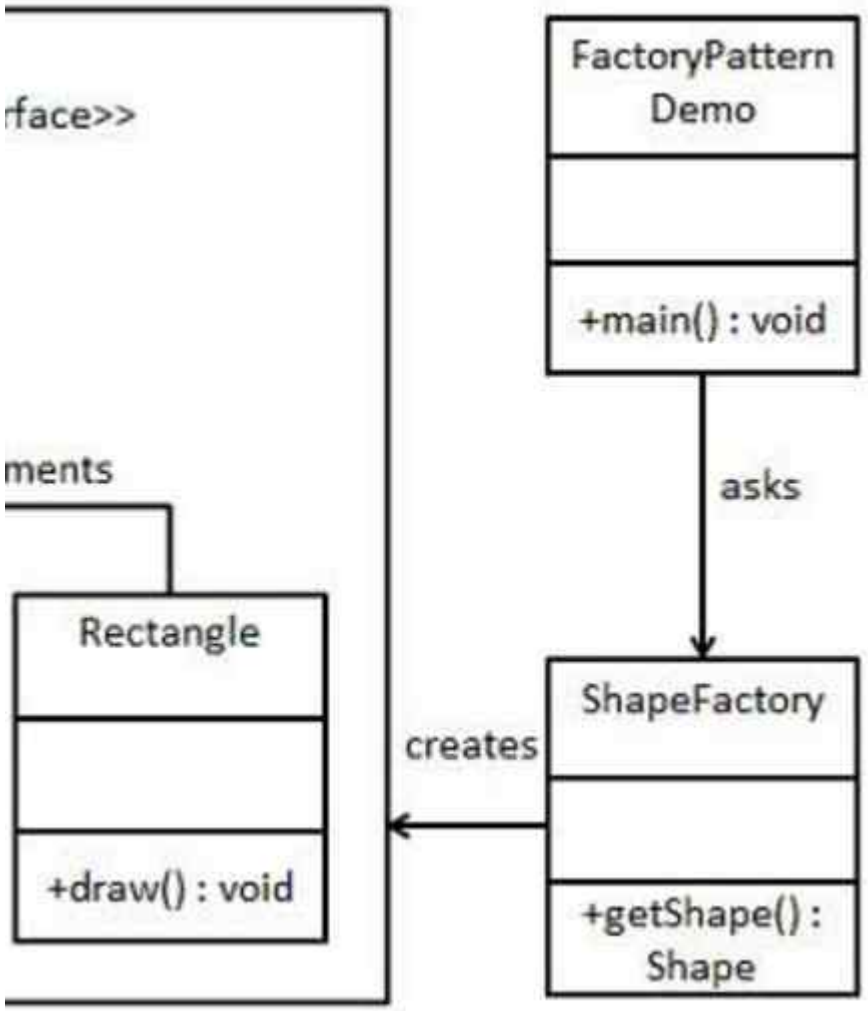
```java
public class Circle implements Shape
{
   @Override
   public void draw() {
      System.out.println("Dentro de Circle::draw() method.");
   }
}
```



Un ejemplo rápido: el Patrón Factory

```
public class Rectangle implements Shape
{
    @Override
    public void draw()
    {
        System.out.println("Dentro de Rectangle::draw() method.");
    }
}


public class Square implements Shape
{
    @Override
    public void draw()
    {
        System.out.println("Dentro de Square::draw() method.");
    }
}
```




```java
public class ShapeFactory
{
   //usa el método getShape para obtener
   //el objeto del tipo correspondiente
   public Shape getShape(String shapeType)
   {
      if(shapeType == null)
      {
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE"))
      {
         return new Circle();

      }
      else if(shapeType.equalsIgnoreCase("RECTANGLE"))
      {
         return new Rectangle();

      }
      else if(shapeType.equalsIgnoreCase("SQUARE"))
      {
         return new Square();
      }
      return null;
   }
}
```

```java
public class FactoryPatternDemo
{
   public static void main(String[] args)
   {
      ShapeFactory shapeFactory = new ShapeFactory();

      //obtiene un objeto de tipo Circle
      //y llama a su método draw().
      Shape shape1 = shapeFactory.getShape("CIRCLE");
      shape1.draw();

      //obtiene un objeto de tipo Rectangle
      //y llama a su método draw().
      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      shape2.draw();

      //obtiene un objeto de tipo Square
      //y llama a su método draw().
      Shape shape3 = shapeFactory.getShape("SQUARE");
      shape3.draw();
   }
}
```

```
Dentro de Circle::draw() method.
Dentro de Rectangle::draw() method.
Dentro de Square::draw() method.
```

Aporte de los Patrones

Que viene “gratis” con los design patterns?

- Experiencia.
- Sus nombres forman, colectivamente, un vocabulario que ayuda a los desarrolladores a comunicarse mejor, a hablar en el mismo idioma.
- Si la documentación usa patrones, simplifica la lectura y comprensión.
- Facilita el rediseño.

Qué NO viene “gratis” con los design patterns?

- La creatividad:
  - Un patrón de diseño no garantiza nada por sí solo. Los patrones no buscan suplantar al humano en el proceso creativo.
  - Necesitamos seguir usando la creatividad para aplicar correctamente los patterns.

El Patrón TEMPLATE METHOD

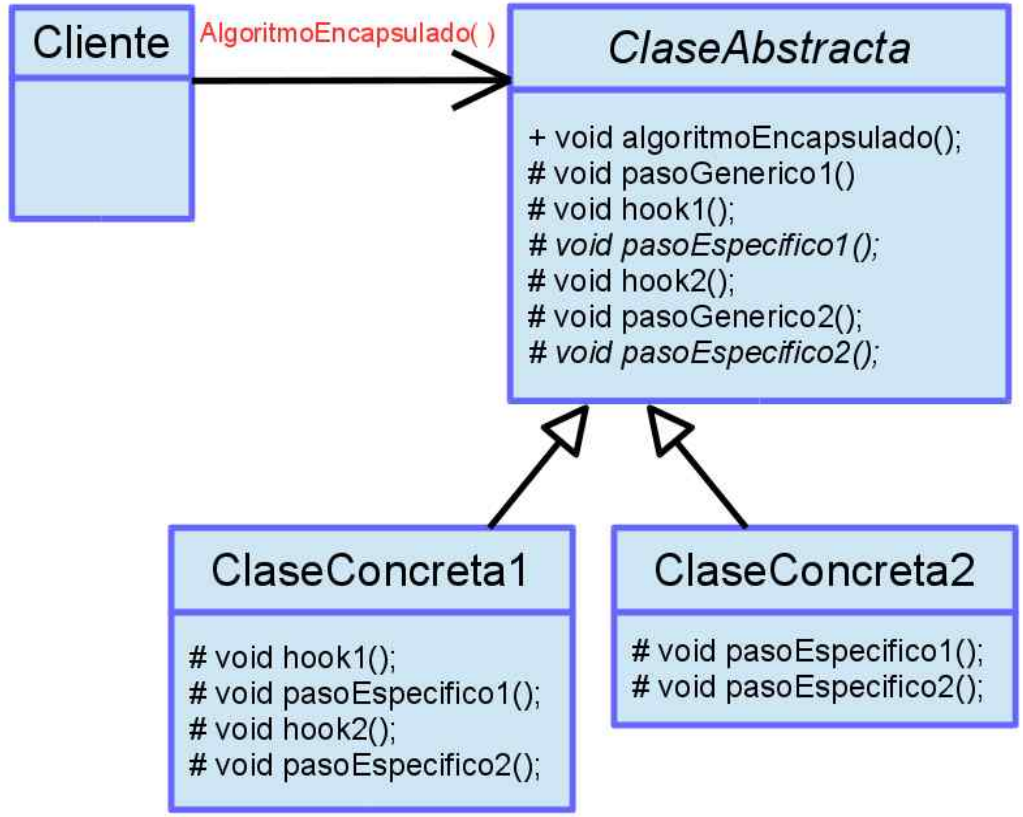
(de comportamiento)

El Patrón TEMPLATE METHOD

Objetivo:

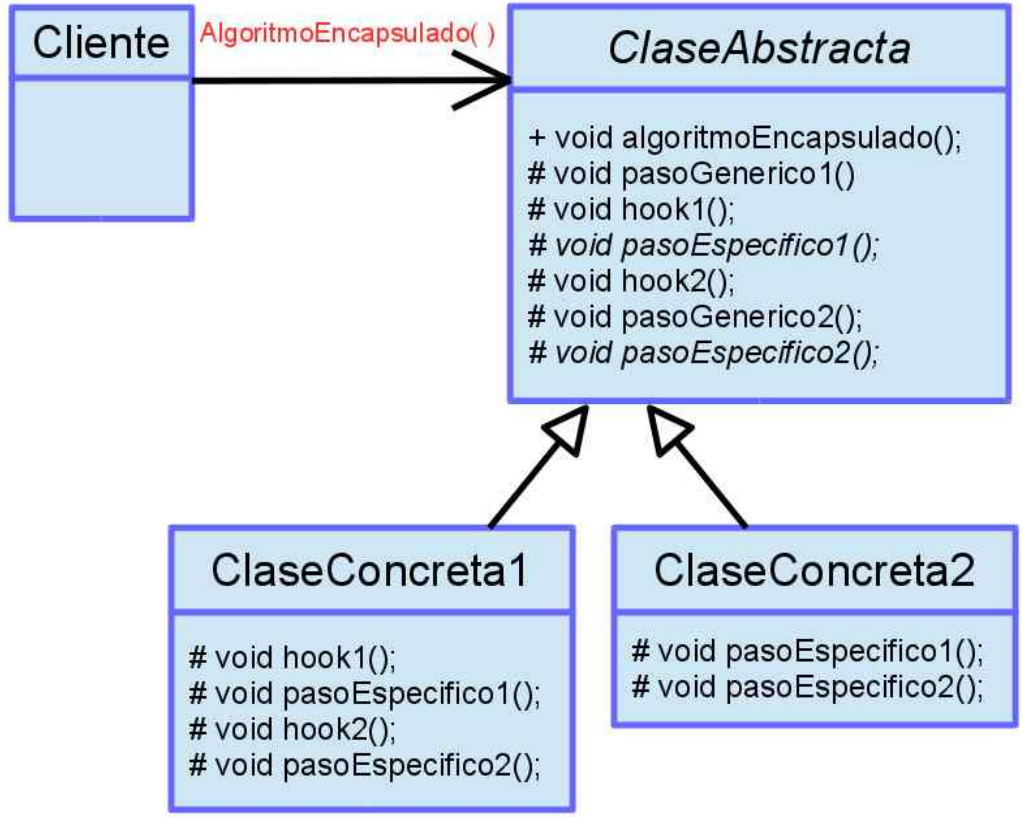
Permitir que ciertos pasos de un algoritmo definidos en un método de una clase sean redefinidos en sus clases derivadas sin necesidad de sobrecargar la operación entera.

Nos permite establece una forma de encapsular algoritmos.



El Patrón TEMPLATE METHOD

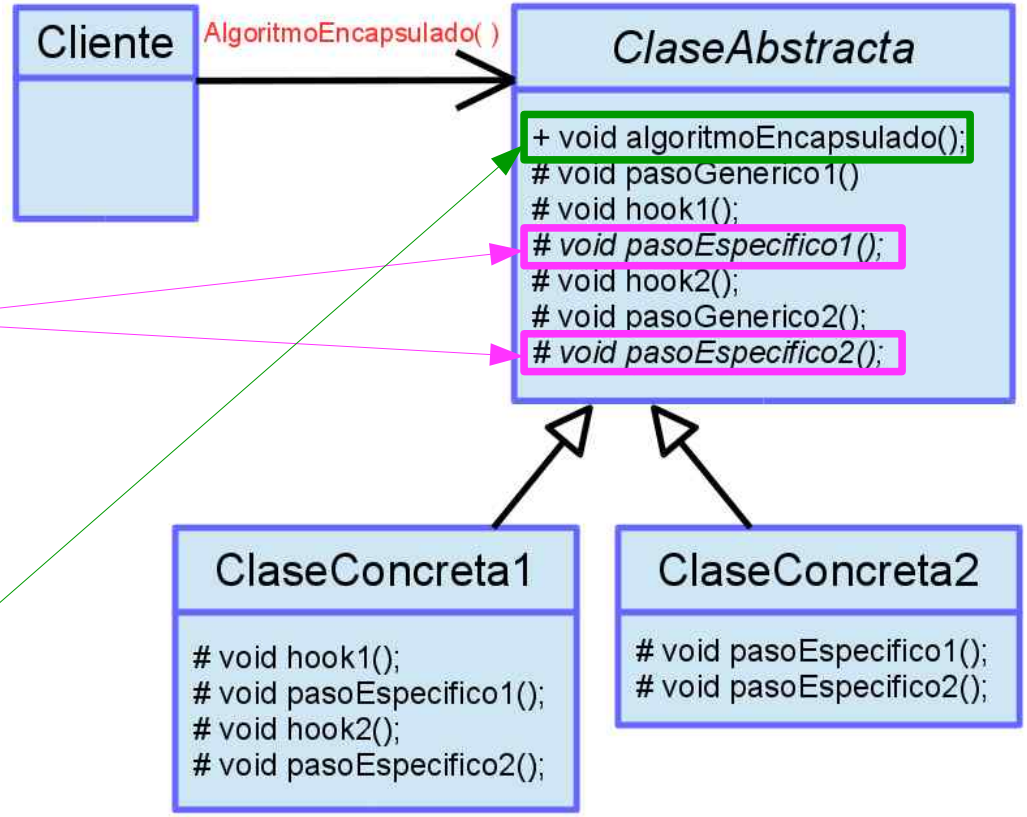
Este patrón se basa en un principio muy sencillo: si un algoritmo puede aplicarse a varios supuestos en los que únicamente *cambie un pequeño número de operaciones*, la idea será utilizar una clase para modelarlo a través de sus operaciones. **Esta clase base se encargará de definir los pasos comunes del algoritmo,** mientras que **las clases que hereden de ella implementarán los detalles propios** de cada caso concreto, es decir, el código específico para cada caso.



El Patrón TEMPLATE METHOD

Procedimiento:

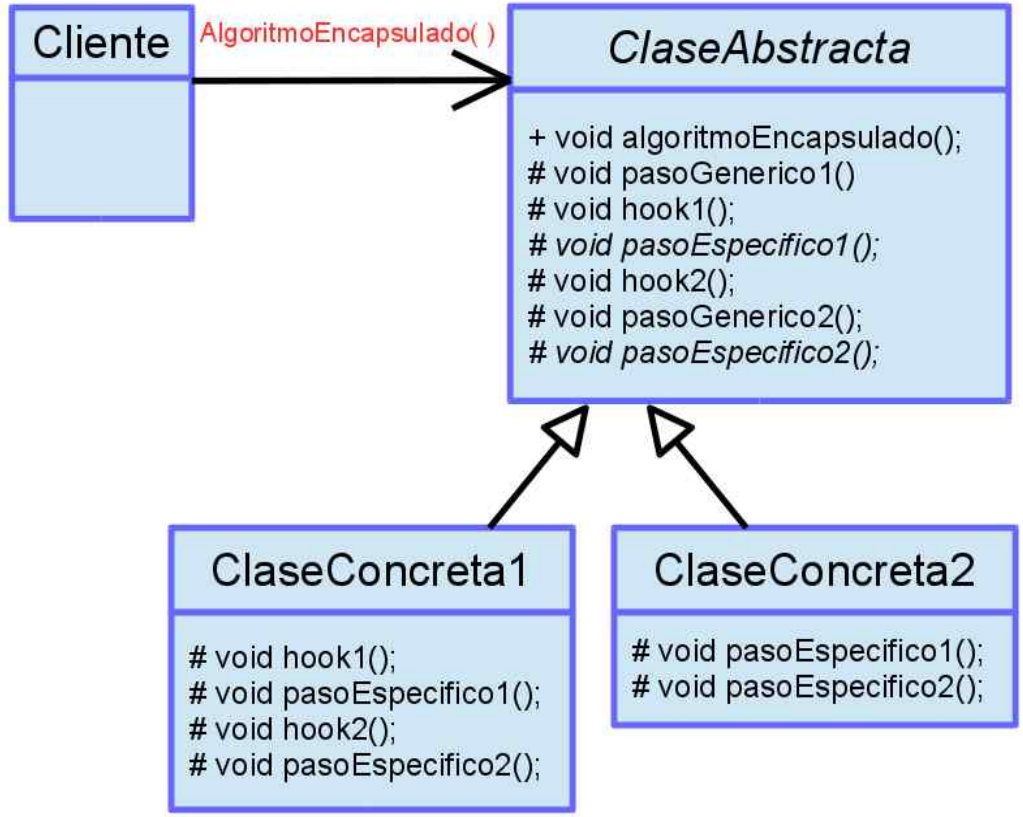
- Se declara una clase abstracta, que será la plantilla o modelo. Esta clase definirá una serie de métodos. Aquellos que sean comunes estarán implementados. Aquellos que dependan de cada caso concreto, se declararán como **abstractos,** obligando a las clases hijas a implementarlos.
- Cada clase derivada implementará los métodos específicos, acudiendo a la clase base para ejecutar el código común.
- La clase base también se encargará de la **lógica del algoritmo**, ejecutando los pasos en un orden preestablecido (las clases hijas no deberían poder modificar el algoritmo, únicamente definir la funcionalidad específica que tienen que implementar).



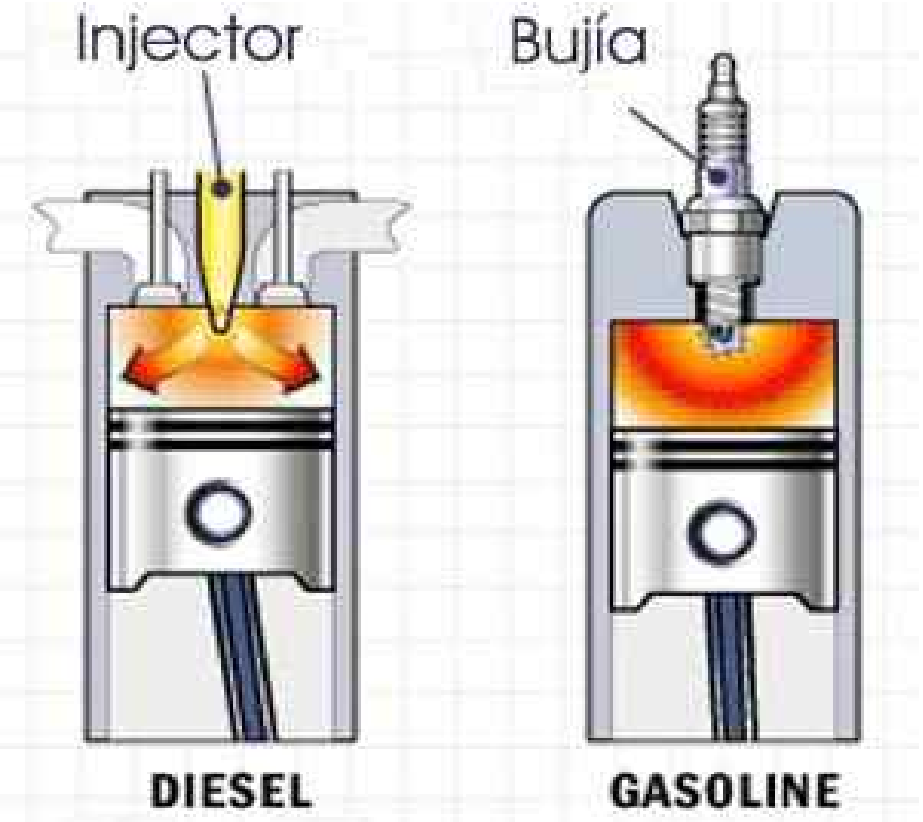
El Patrón TEMPLATE METHOD

Inversión de dependencias:

Dado que la clase padre es la que se encarga de llamar los métodos de las clases derivadas (los pasos del algoritmo estarán implementado en la clase base), se trata de una aplicación manifiesta del *principio de inversión de dependencias: la clase base no tiene por qué saber nada acerca de sus hijas, pero aún así, se encargará de invocar su funcionalidad cuando sea necesario.*



Ejemplo: Motor de 4 tiempos



Veremos dos formas de implementar el Patrón: con gancho y sin gancho

Ejemplo: Motor de 4 tiempos

Nafta

1. Admisión: el descenso del pistón crea un vacío que aspira la mezcla de aire y combustible de la válvula de admisión. La válvula de escape permanece cerrada.
2. Compresión: una vez que el pistón ha bajado hasta el final, se cierra la válvula de admisión. El pistón asciende, comprimiendo la mezcla y aumentando la presión.
3. Explosión: el pistón alcanza la parte superior y la bujía produce una chispa que hace explotar la mezcla de aire y combustible, haciendo que el pistón vuelva a descender.
4. Escape: la válvula de escape se abre. El pistón asciende nuevamente, empujando los gases resultantes de la explosión y comenzando un nuevo ciclo.

Diesel

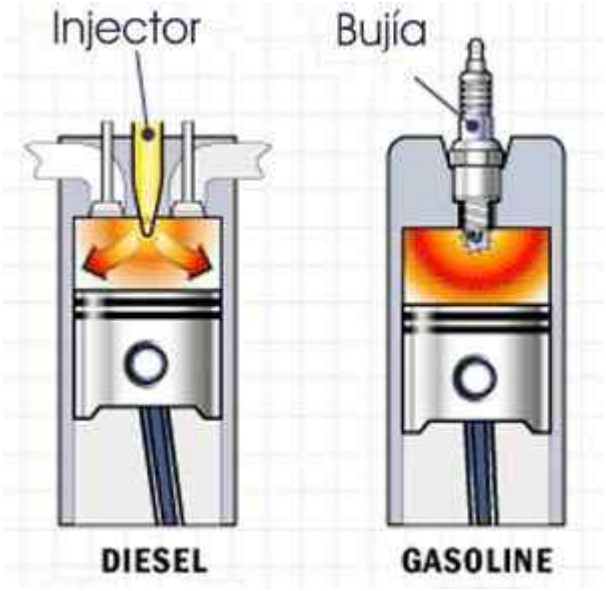
1. Admisión: el descenso del pistón crea un vacío que aspira aire desde la válvula de admisión. La válvula de escape permanece cerrada.
2. Compresión: una vez que el pistón ha bajado hasta el final, se cierra la válvula de admisión. El pistón asciende, comprimiendo el aire y aumentando la presión.
3. Combustión: los inyectores pulverizan el combustible, haciendo que la presión se encargue de aumentar la temperatura, haciendo que se produzca la combustión y la expansión de los gases que fuerzan el descenso del pistón.
4. Escape: la válvula de escape se abre. El pistón asciende nuevamente, empujando los gases resultantes de la explosión y comenzando un nuevo ciclo.

Ambos motores tienen un funcionamiento muy similar. Las fases 2 y 4 (compresión y escape) son idénticas, la fase 1 (admisión) varía ligeramente, mientras que la fase 3 (explosión en el motor de gasolina, combustión en el motor diesel) tiene un comportamiento diferente. ¿Cómo encaja aquí el patrón Template Method?

Ejemplo: Motor de 4 tiempos

superclase Motor

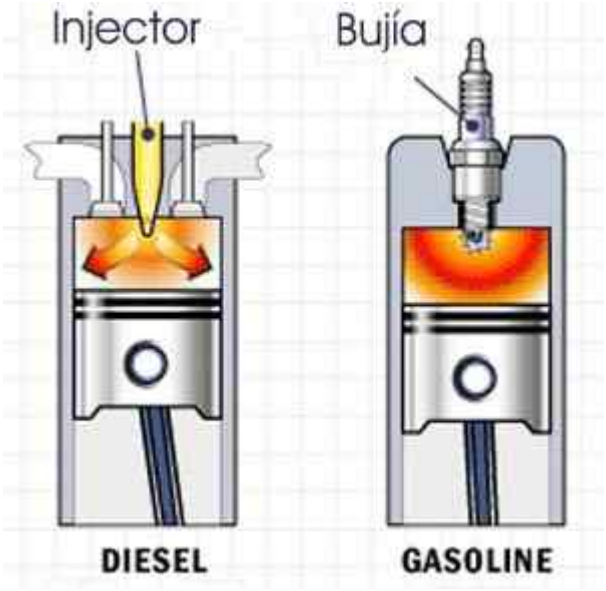
Dado que el algoritmo para realizar el ciclo del motor tiene los mismos pasos efectuados en el mismo orden, es el contexto adecuado para utilizar este patrón. Podemos crear una superclase Motor que implemente las fases comunes a ambos motores (Compresión, Escape) más el algoritmo RealizarFaseMotor, que será el encargado de invocar los métodos en un orden fijo. Esta ejecución será invariable, por lo que las clases derivadas únicamente podrán (y deberán) implementar las partes específicas de cada motor.

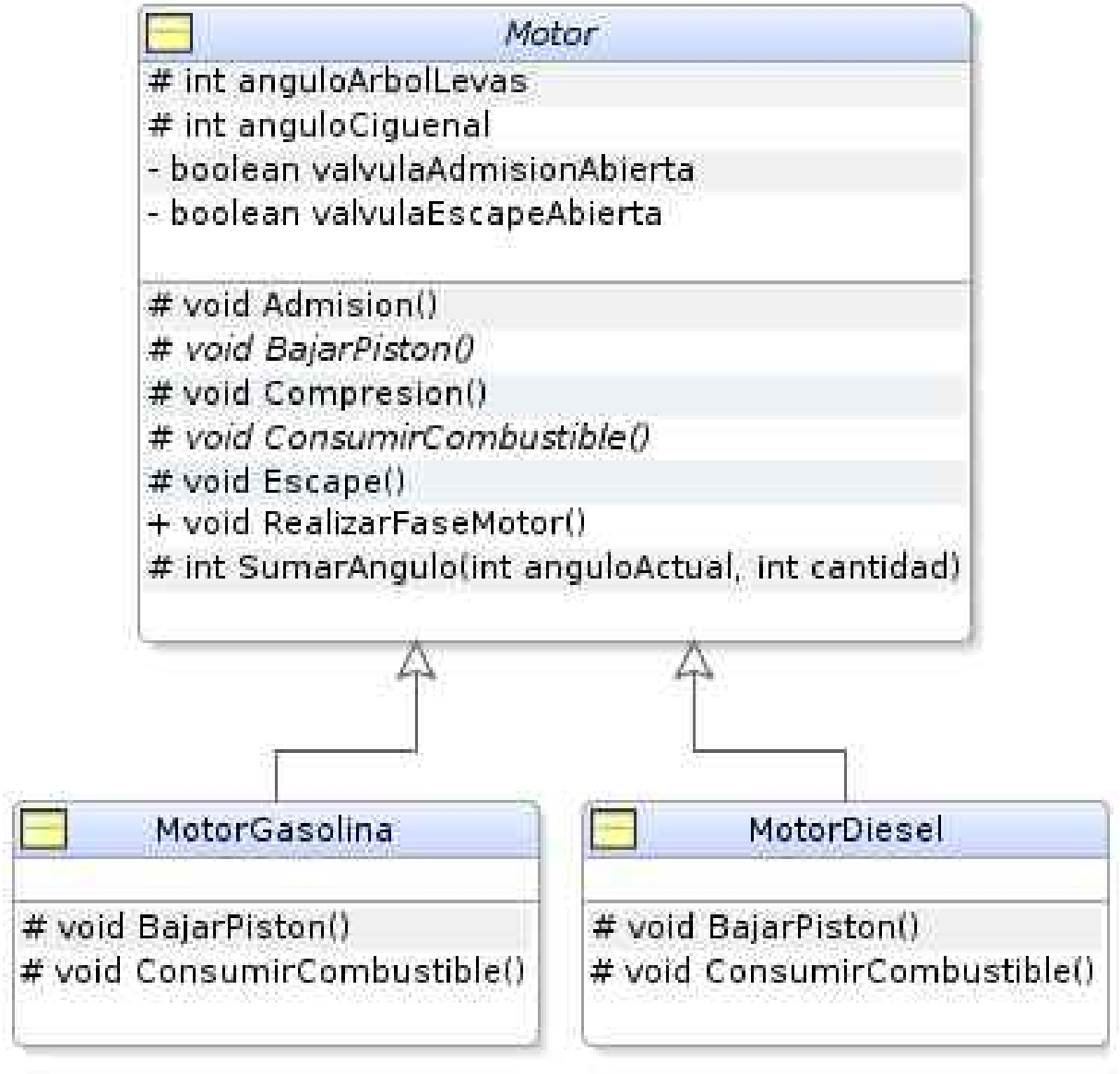


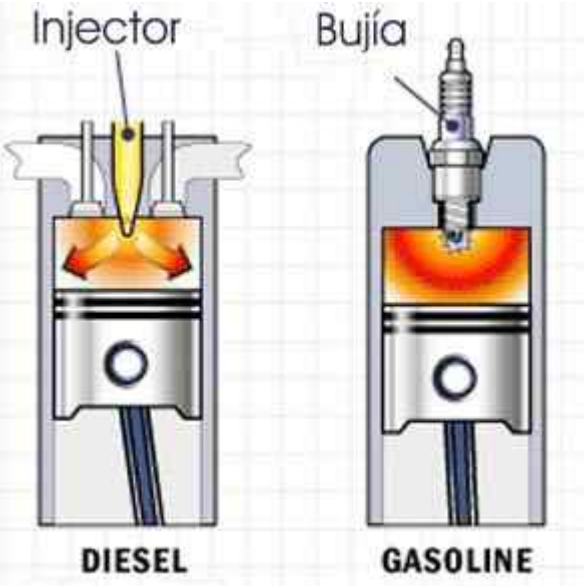
Ejemplo: Motor de 4 tiempos

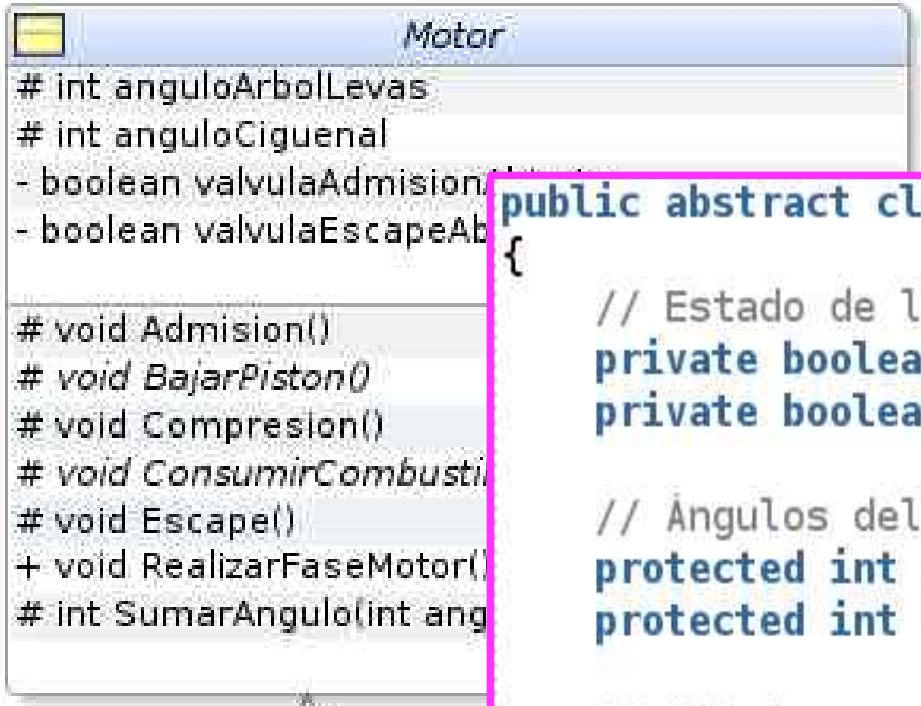
encapsular aquello que es susceptible de cambiar:

Dado que la fase Admisión varía ligeramente (inyección de mezcla de combustible y gas en el motor de gasolina, frente a inyección de gas en el motor diesel), es posible implementar la parte común en el propio método de la superclase y encapsular únicamente la parte que varía en otro método que será invocado desde Compresión. Así respetaremos otro de los principios de la orientación a objetos: encapsular aquello que es susceptible de cambiar.




Motor
int anguloArbolLevas
int anguloCiguenal
- boolean valvulaAdmisionAbierta
- boolean valvulaEscapeAbierta
void Admision()
void BajarPiston()
void Compresion()
void ConsumirCombustible()
void Escape()
+ void RealizarFaseMotor()
# int SumarAngulo(int anguloActual, int cantidad)
MotorGasolina
void BajarPiston()
void ConsumirCombustible()
MotorDiesel
void BajarPiston()
void ConsumirCombustible()


Injector
Bujía
DIESEL
GASOLINE




```java
public abstract class Motor
{
    // Estado de las válvulas
    private boolean valvulaAdmisionAbierta = false;
    private boolean valvulaEscapeAbierta = false;

    // Ángulos del cigueñal y del árbol de levas
    protected int anguloCiguenal = 0;
    protected int anguloArbolLevas = 0;

    // Método que mantendrá el ángulo entre 0 y 359 grados
    protected int sumarAngulo(int anguloActual, int cantidad)
    {
        if (anguloActual + cantidad >= 360)
            return anguloActual + cantidad - 360;
        else
            return anguloActual + cantidad;
    }
```

```java
// Segunda Fase: Compresión
protected void compresion()
{
    System.out.println("COMENZANDO FASE DE COMPRESION");
    // Se cierra la válvula de admisión
    valvulaAdmisionAbierta = false;
    // Giros del cigueñal y del árbol de levas
    anguloCiguenal = sumarAngulo(anguloCiguenal, 360);
    anguloArbolLevas = sumarAngulo(anguloArbolLevas, 180);
    System.out.println("Angulo del ciguenal: " + anguloCiguenal);
    System.out.println("Angulo del arbol de levas: " + anguloArbolLevas);
    System.out.println("Valvula de admision abierta: " + valvulaAdmisionAbierta);
    System.out.println("Valvula de escape abierta: " + valvulaEscapeAbierta + "\n");
}

// Cuarta Fase: Escape
protected void escape()
{
    System.out.println("COMENZANDO FASE DE ESCAPE");
    // Se abre la válvula de escape
    valvulaEscapeAbierta = true;
    // Giros del cigueñal y del árbol de levas
    anguloCiguenal = sumarAngulo(anguloCiguenal, 180);
    anguloArbolLevas = sumarAngulo(anguloArbolLevas, 90);
    System.out.println("Angulo del ciguenal: " + anguloCiguenal);
    System.out.println("Angulo del arbol de levas: " + anguloArbolLevas);
    System.out.println("Gases expulsados. Fin de ciclo");
}
```

```java
// Tercera Fase: Consumo del combustible. Dado que depende del motor concreto,
// este método será abstracto y deberá ser implementado por la clase derivada.
protected abstract void consumirCombustible();

// La bajada del pistón depende del motor concreto, por lo que deberá ser implementada
// por la clase hija.
protected abstract void bajarPiston();

// Primera Fase: Admisión
protected void admision()
{
    System.out.println("COMENZANDO FASE DE ADMISION");
    // Se abre la válvula de admisión y se cierra la válvula de escape
    valvulaAdmisionAbierta = true;
    valvulaEscapeAbierta = false;
    // Se baja el pistón. Esta operación será distinta en el motor diesel (que
    // inyectará aire) o gasolina (que inyectará una mezcla de aire y combustible)
    bajarPiston();
    anguloCiguenal = sumarAngulo(anguloCiguenal, 180);
    anguloArbolLevas = sumarAngulo(anguloArbolLevas, 90);
    System.out.println("Angulo del ciguenal: " + anguloCiguenal);
    System.out.println("Angulo del arbol de levas: " + anguloArbolLevas);
    System.out.println("Valvula de admision abierta: " + valvulaAdmisionAbierta);
    System.out.println("Valvula de escape abierta: " + valvulaEscapeAbierta + "\n");
}
```

```java
public class MotorGasolina extends Motor
{
    @Override
    protected void consumirCombustible()
    {
        System.out.println("Inyectando aire y combustible en el motor");
    }

    @Override
    protected void bajarPiston()
    {
        System.out.println("COMENZANDO FASE DE EXPLOSIÓN");

        System.out.println("Iniciando chispa en la bujía");
        System.out.println("La explosión provoca el movimiento del pistón");

        anguloCiguenal = sumarAngulo(anguloCiguenal, 180);
        anguloArbolLevas = sumarAngulo(anguloArbolLevas, 90);

        System.out.println("Angulo del ciguenal: " + anguloCiguenal);
        System.out.println("Angulo del arbol de levas: " + anguloArbolLevas + "\n");
    }
}
```

```java
public class MotorDiesel extends Motor
{
    @Override
    protected void consumirCombustible()
    {
        System.out.println("Inyectando aire en el motor");
    }

    @Override
    protected void bajarPiston()
    {
        System.out.println("COMENZANDO FASE DE COMBUSTIÓN");

        System.out.println("Inyectando combustible pulverizado en el motor");
        System.out.println("La presión provoca el movimiento del pistón");

        anguloCiguenal = sumarAngulo(anguloCiguenal, 180);
        anguloArbolLevas = sumarAngulo(anguloArbolLevas, 90);

        System.out.println("Angulo del ciguenal: " + anguloCiguenal);
        System.out.println("Angulo del arbol de levas: " + anguloArbolLevas + "\n");
    }
}
```

Las dos diferencias entre ambos radican en la compresión (inyección de mezcla en el motor gasolina por inyección de aire en el motor diesel) y en el consumo de combustible (explosión en el motor gasolina por combustión en el motor diesel). Salvo esto, el algoritmo del motor es exactamente el mismo, y viene determinado por el método público **`realizarFaseMotor()`** codificado en la clase **`Motor`**.

```java
// Método público que ejecutará el algoritmo completo
public void realizarFaseMotor()
{
    admision(); // Parcialmente implementado en la clase base
    compresion(); // Implementado en la clase base
    consumirCombustible(); // Delegado en las clases hijas
    escape(); // Implementado en la clase base
}
```

```
public static void main(String[] args)
{
    Motor mGasolina = new MotorGasolina();
    Motor mDiesel = new MotorDiesel();
    mGasolina.realizarFaseMotor();
    System.out.println("--------------------
    mDiesel.realizarFaseMotor();
}
```

De hecho, si usamos el método para comprobar el funcionamiento de ambos motores, veremos el resultado siguiente:

COMENZANDO FASE DE ADMISION
COMENZANDO FASE DE EXPLOSIÓN
Iniciando chispa en la bujía
La explosión provoca el movimiento del pistón
Angulo del ciguenal: 180
Angulo del arbol de levas: 90

Angulo del ciguenal: 0
Angulo del arbol de levas: 180
Valvula de admision abierta: true
Valvula de escape abierta: false

COMENZANDO FASE DE COMPRESION
Angulo del ciguenal: 0
Angulo del arbol de levas: 0
Valvula de admision abierta: false
Valvula de escape abierta: false

Inyectando aire y combustible en el motor
COMENZANDO FASE DE ESCAPE
Angulo del ciguenal: 180
Angulo del arbol de levas: 90
Gases expulsados. Fin de ciclo

COMENZANDO FASE DE ADMISION
COMENZANDO FASE DE COMBUSTIÓN
Inyectando combustible pulverizado en el motor
La presión provoca el movimiento del pistón
Angulo del ciguenal: 180
Angulo del arbol de levas: 90

Angulo del ciguenal: 0
Angulo del arbol de levas: 180
Valvula de admision abierta: true
Valvula de escape abierta: false

COMENZANDO FASE DE COMPRESION
Angulo del ciguenal: 0
Angulo del arbol de levas: 0
Valvula de admision abierta: false
Valvula de escape abierta: false

Inyectando aire en el motor
COMENZANDO FASE DE ESCAPE
Angulo del ciguenal: 180
Angulo del arbol de levas: 90
Gases expulsados. Fin de ciclo
Process exited with exit code 0.

Ejemplo: Motor de 4 tiempos

Algunas conclusiones del ejemplo:

Sencillo → Mantener encapsulado e inmutable el cuerpo del algoritmo evita efectos indeseados

Rígido → Limita muchísimo la posibilidad de añadir pasos intermedios

Ejemplo: Motor de 4 tiempos

qué ocurriría si nuestro motor fuera turbo?
El turbo realiza una compresión del aire antes de introducirlo en la cámara de explosión. Sin embargo, este paso estaría justo antes de la admisión.

1. Habrá motores que implementen turbo y que no lo implementen?.
2. Creamos por tanto una clase MotorTurboGasolina y MotorTurboDiesel?
3. Hacemos también abstracto el método admisión() para que este funcionamiento pueda personalizarse en las clases hijas?

No es necesario. Para este tipo de situaciones, el patrón proporciona lo que se conocen como Hooks, métodos de enganche o ganchos.

Ejemplo: Motor de 4 tiempos

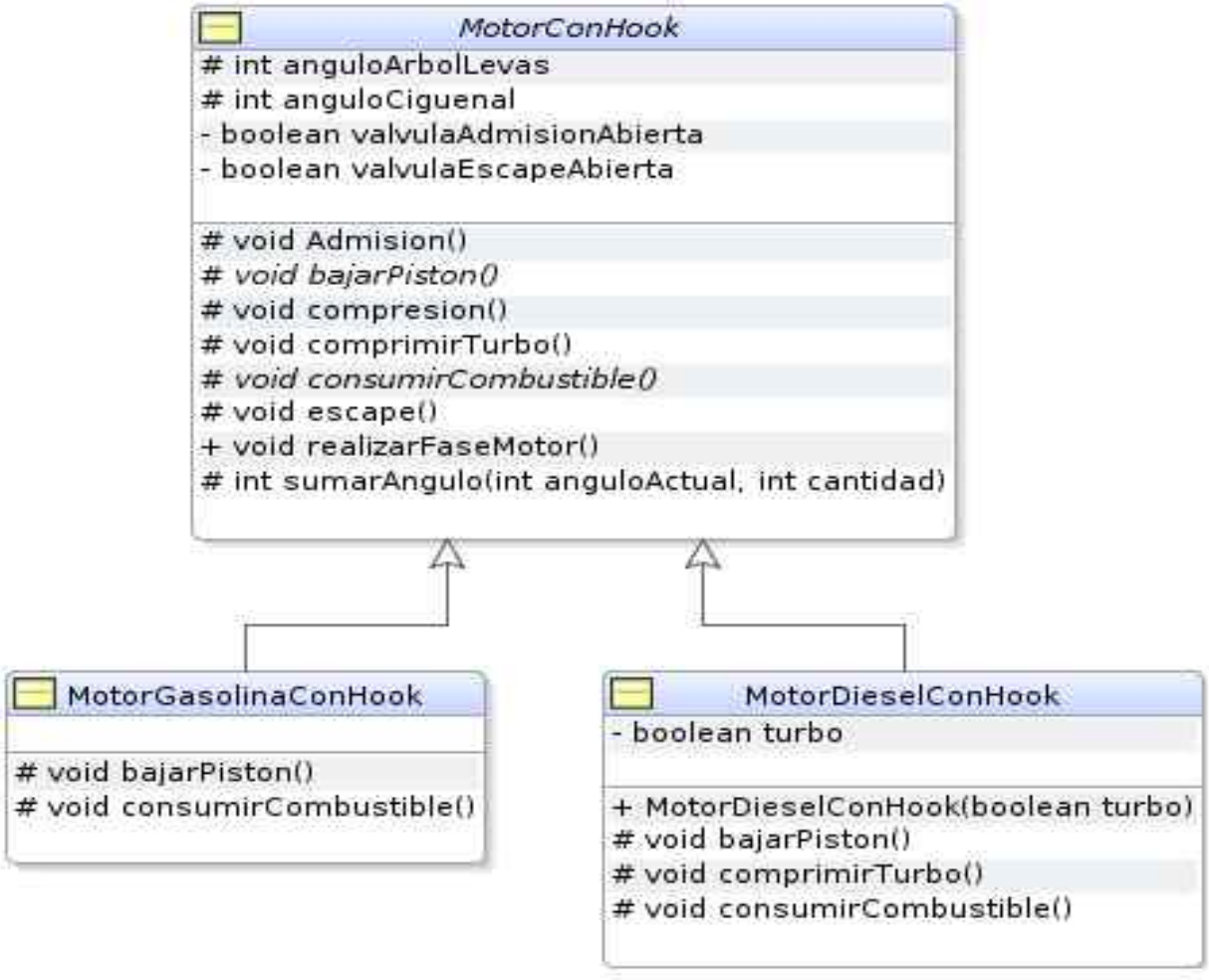
Algoritmos con gancho

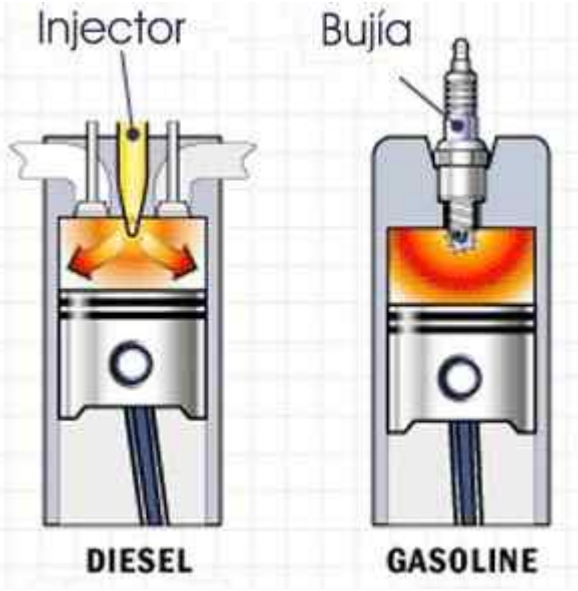
Se implementan en la clase base con un *comportamiento por defecto*, que puede ser sobrecargado en las clases derivadas o ejecutado como está.

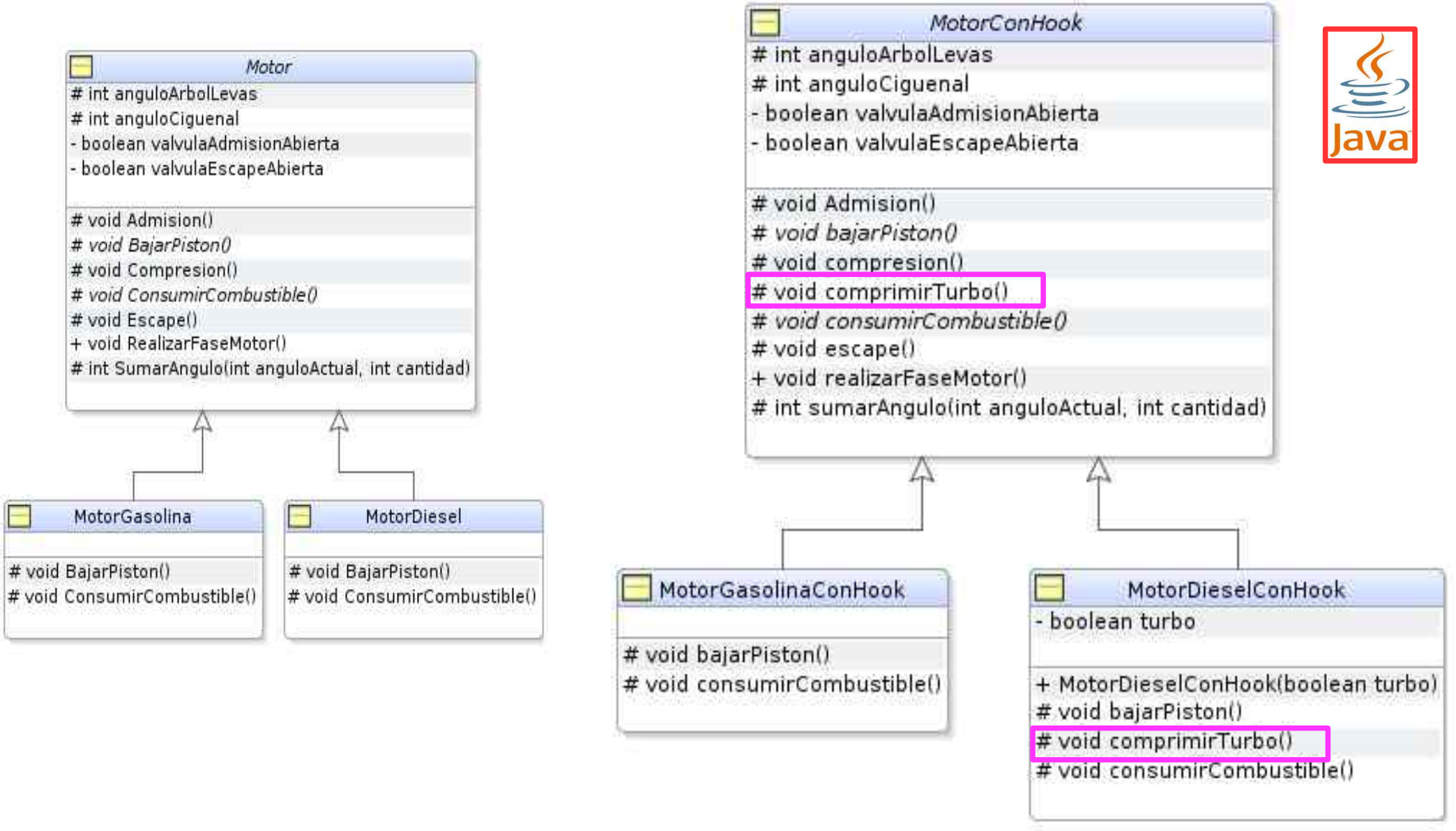
De este modo, se proporcionan pasos opcionales, haciendo que las clases derivadas enganchen funcionalidad opcional en ciertos puntos del algoritmo.

Estos hooks **suelen utilizarse normalmente con cláusulas condicionales**, de forma que se ejecuten en ciertas condiciones concretas.

Ejemplo: Motor de 4 tiempos






Motor
int anguloArbolLevas
int anguloCiguenal
- boolean valvulaAdmisionAbierta
- boolean valvulaEscapeAbierta
void Admision()
void BajarPiston()
void Compresion()
void ConsumirCombustible()
void Escape()
+ void RealizarFaseMotor()
# int SumarAngulo(int anguloActual, int cantidad)
MotorGasolina
void BajarPiston()
void ConsumirCombustible()
MotorDiesel
void BajarPiston()
void ConsumirCombustible()
MotorConHook
int anguloArbolLevas
int anguloCiguenal
- boolean valvulaAdmisionAbierta
- boolean valvulaEscapeAbierta
void Admision()
void bajarPiston()
void compresion()
void comprimirTurbo()
void consumirCombustible()
void escape()
+ void realizarFaseMotor()
# int sumarAngulo(int anguloActual, int cantidad)
Java
MotorGasolinaConHook
void bajarPiston()
void consumirCombustible()
MotorDieselConHook
- boolean turbo
+ MotorDieselConHook(boolean turbo)
# void bajarPiston()
# void comprimirTurbo()
# void consumirCombustible()

MotorConHook

```java
// método base que puede ser sobreescrito
protected void comprimirTurbo()
{
    System.out.println("Turbo no presente");
}

// Método público que ejecutará el algoritmo completo
public void realizarFaseMotor()
{
    comprimirTurbo();          // Hook (método opcional)
    Admision();                // Parcialmente implementado en la clase base
    compresion();              // Implementado en la clase base
    consumirCombustible();     // Delegado en las clases hijas
    escape();                  // Implementado en la clase base
}
```

MotorDieselConHook

```java
public class MotorDieselConHook extends MotorConHook
{
    private boolean turbo = false;

    public MotorDieselConHook(boolean turbo)
    {
        this.turbo = turbo;
    }

    protected void comprimirTurbo()
    {
        // Si el coche es turbo, ejecutará su propio código. En caso contrario, efectuará
        // la operación por defecto
        if (turbo)
            System.out.println("Comprimiendo aire en el turbo antes de la admisión");
        else
            super.comprimirTurbo();
    }
```

El motor gasolina no será modificado, de modo que ejecutará el método por defecto (no se “enganchará” al método ComprimirTurbo). Si ahora modificamos nuestro programa para instanciar un motor diesel turbo de la siguiente manera:

```
public class Prueba
{
    public static void main(String[] args)
    {
        MotorConHook mGasolina = new MotorGasolinaConHook();
        MotorConHook mDiesel = new MotorDieselConHook(true);
        mGasolina.realizarFaseMotor();
        System.out.println("--------------------------------
        mDiesel.realizarFaseMotor();
    }
}
```

Veremos que el motor diesel hace uso del gancho para añadir su propia funcionalidad, mientras que el motor gasolina ha delegado en la clase base el funcionamiento por defecto:

Turbo no presente

COMENZANDO FASE DE ADMISION
COMENZANDO FASE DE EXPLOSIÓN
Iniciando chispa en la bujía
La explosión provoca el movimiento del pistón
Angulo del ciguenal: 180
Angulo del arbol de levas: 90

Angulo del ci
Angulo del arbol de levas: 180
Valvula de admision abierta: true
Valvula de escape abierta: false

COMENZANDO FASE DE COMPRESION
Angulo del ciguenal: 0
Angulo del arbol de levas: 0
Valvula de admision abierta: false
Valvula de escape abierta: false

Inyectando aire y combustible en el motor
COMENZANDO FASE DE ESCAPE
Angulo del ciguenal: 180
Angulo del arbol de levas: 90
Gases expulsados. Fin de ciclo

MotorConHook.comprimirTurbo();

Comprimiendo aire en el turbo antes de la admisión

COMENZANDO FASE DE ADMISION
Inyectando aire en el motor
Angulo del ciguenal: 180
Angulo del arbol de levas: 90
Valvula de admision abierta: true
Valvula de escape abierta: false

MENZANDO FASE DE COMPRESION
Angulo del ciguenal: 180
Angulo del arbol de levas: 270
Valvula de admision abierta: false
Valvula de escape abierta: false

COMENZANDO FASE DE COMBUSTIÓN
Inyectando combustible pulverizado en el motor
La presión provoca el movimiento del pistón
Angulo del ciguenal: 0
Angulo del arbol de levas: 0

COMENZANDO FASE DE ESCAPE
Angulo del ciguenal: 180
Angulo del arbol de levas: 90
Gases expulsados. Fin de ciclo
Process exited with exit code 0.

El Patrón TEMPLATE METHOD

¿Cuándo utilizar este patrón?

Este patrón es aconsejable en los siguientes supuestos:

- Cuando se cuenta con un algoritmo aplicable a varias situaciones, cuya implementación difiere únicamente en algunos pasos.
- Arquitecturas donde los pasos de un proceso estén definidos (el qué), pero sea necesario establecer los detalles sobre cómo realizarlos (el cómo).
- Módulos en los que exista una gran cantidad de código duplicado que pueda ser factorizado en pasos comunes.