Reuso del desarrollo a nivel arquitectural

M. Telleria, L. Barros, J.M. Drake

Patrones de diseño

Soluciones de **diseño** que son válidas en distintos contextos y que han sido aplicadas con éxito en otras ocasiones.

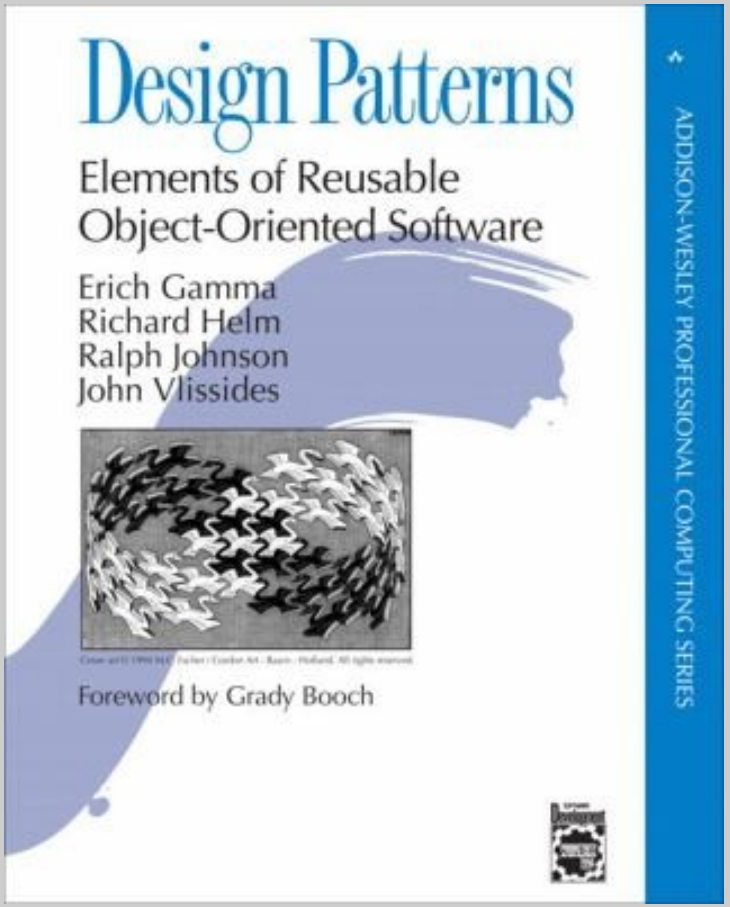
- Se debe haber comprobado su **efectividad** resolviendo problemas similares en ocasiones anteriores.
- Deben ser **reusables**: se puede aplicar a diferentes problemas de diseño en distintas circunstancias.

- Estamos a nivel de UML
- Término acuñado por **Christophe Alexander** en arquitectura (de edificios) y popularizado en informática por el *Gang of Four*:
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

  famosos por el libro *"Design Patterns: Elements of Reusable Object-Oriented Software (1996)"*

Libro del Gang Of Four (GOF book)

Fuente: http://en.wikipedia.org/wiki/Design_Patterns



- Libro de ingeniería de software muy influyente publicado en 1995.
- Neutral respecto al lenguaje. Asume sólo que es orientado a objetos
- Consejos que da:
  - Composición frente a herencia.
  - Centrarse en interfaces frente a implementación.
  - Uso de tipos genéricos.
- Presenta patrones diversos:

  Creación de objetos: Abstract Factory, Builder...
  - Estructura: Adapter, Facade...
  - Comportamiento: Iterator, Interpreter, State...

Ventajas de los patrones de diseño

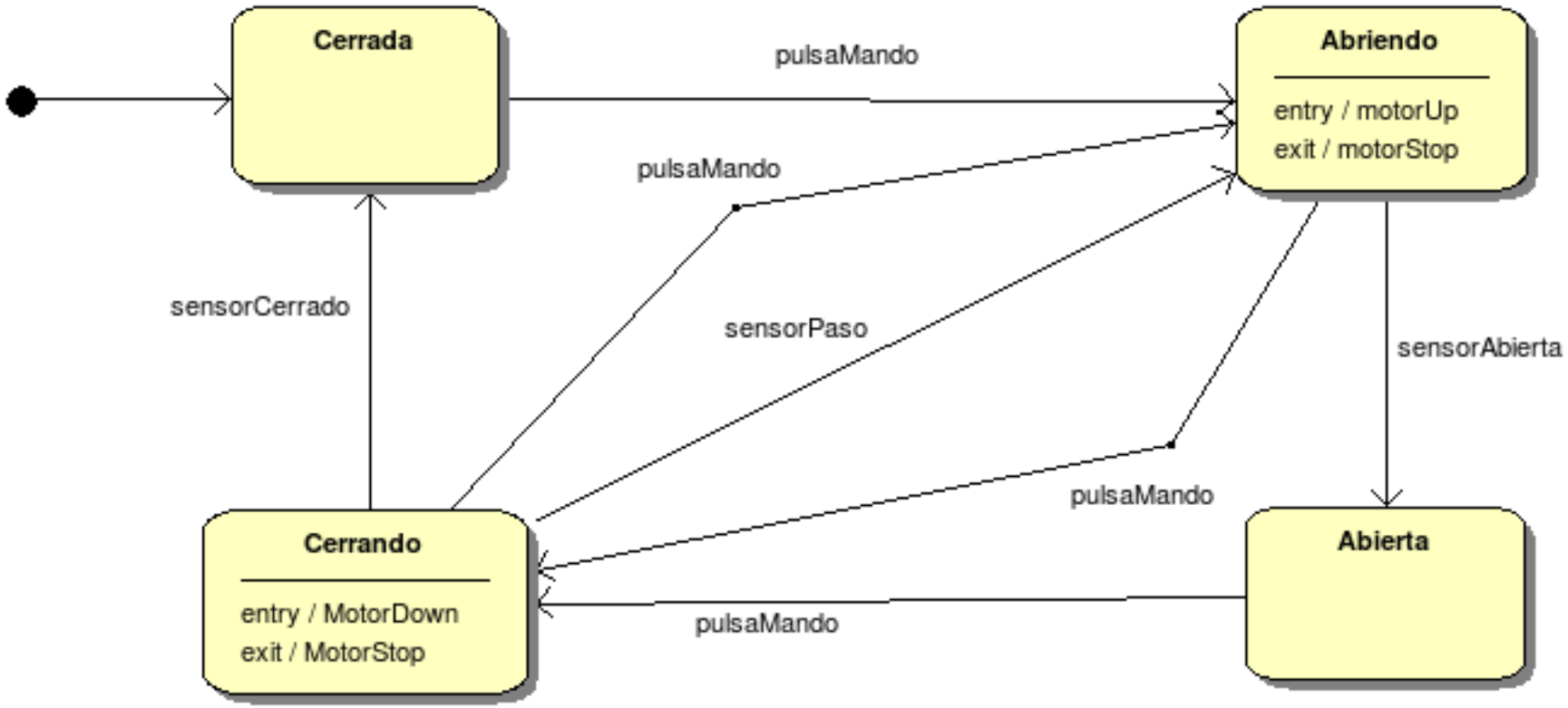
Ayudan a “arrancar” en el diseño de un programa complejo.

- Dan una descomposición de objetos inicial “bien pensada”.
- Pensados para que el programa sea escalable y fácil de mantener.
- Otra gente los ha usado y les ha ido bien.

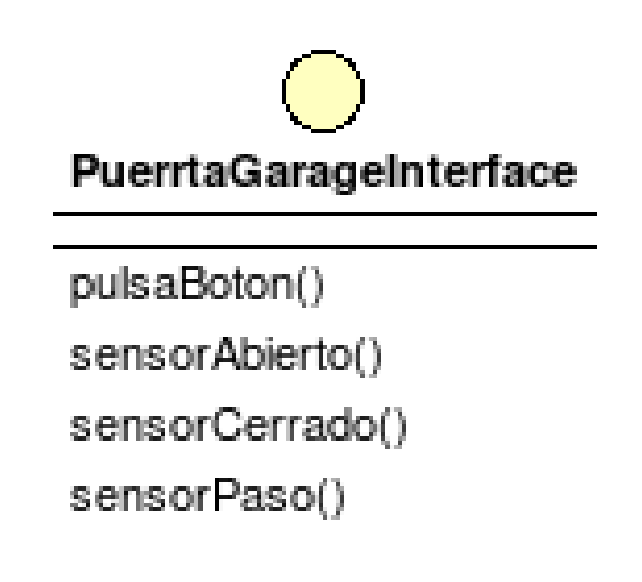
- Ayudan a reutilizar técnicas.
  - Mucha gente los conoce y ya sabe como aplicarlos.
  - Estamos en un alto nivel de abstracción.
  - El diseño se puede aplicar a diferentes situaciones.

Ejemplo: Maquina de estado

Tenemos la siguiente máquina de estado



Objetivo: Ofrecer una interfaz



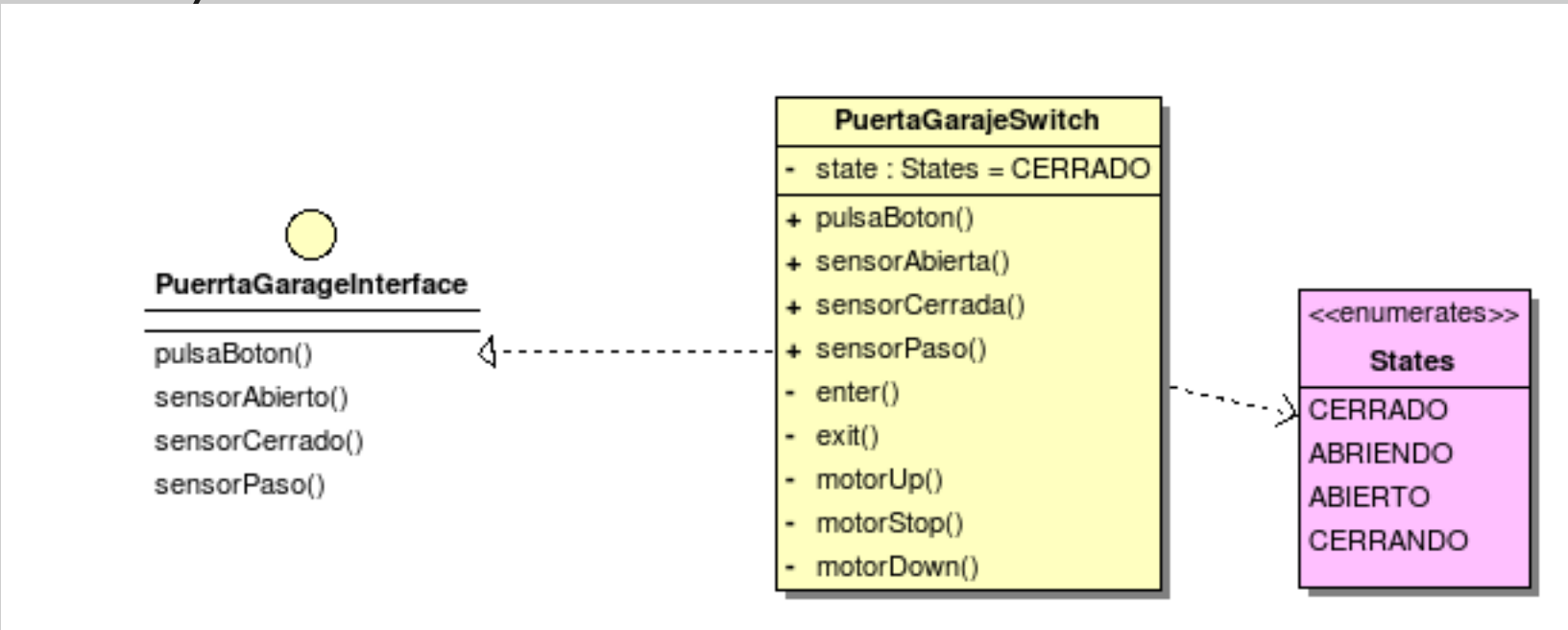

```
public static void main(String[] args) {
    PuertaGarageInt puerta = new PuertaGarage();

    puerta.pulsaBoton();

    puerta.sensorAbierto();

    puerta.pulsaBoton();

    puerta.sensorPaso();

    puerta.sensorAbierto();

    puerta.pulsaBoton();

    puerta.sensorCerrado();
}
```

Compuesta por los eventos externos.

- Con datos estrictamente funcionales

Todos sabemos implementar una m.d.e.

Implementación típica (seguramente la primera que se nos ocurre)



Implementacion clásica (codigo java)

```
public void pulsaBoton(){
  switch(currentState)
  {
  case CERRADO:
    exitAction();
    currentState=GarageState.ABRIENDO;
    entryAction();
    break;
  case ABRIENDO:
    exitAction();
    currentState=GarageState.CERRANDO;
    entryAction();
    break;
  case ABIERTO:
    exitAction();
    currentState=GarageState.CERRANDO;
    entryAction();
    break;
  case CERRANDO:
    exitAction();
    currentState=GarageState.ABRIENDO;
    entryAction();
    break;
  }
}
```

```
private void entryAction()
{
  switch(currentState)
  {
  case ABRIENDO:
    motorUp();
    break;
  case CERRANDO:
    motorDown();
    break;
  default: {};
  }
}

private void exitAction()
{
  switch(currentState){
  case ABRIENDO:
    motorStop();
    break;
  case CERRANDO:
    motorStop();
    break;
  default: {};
}
```

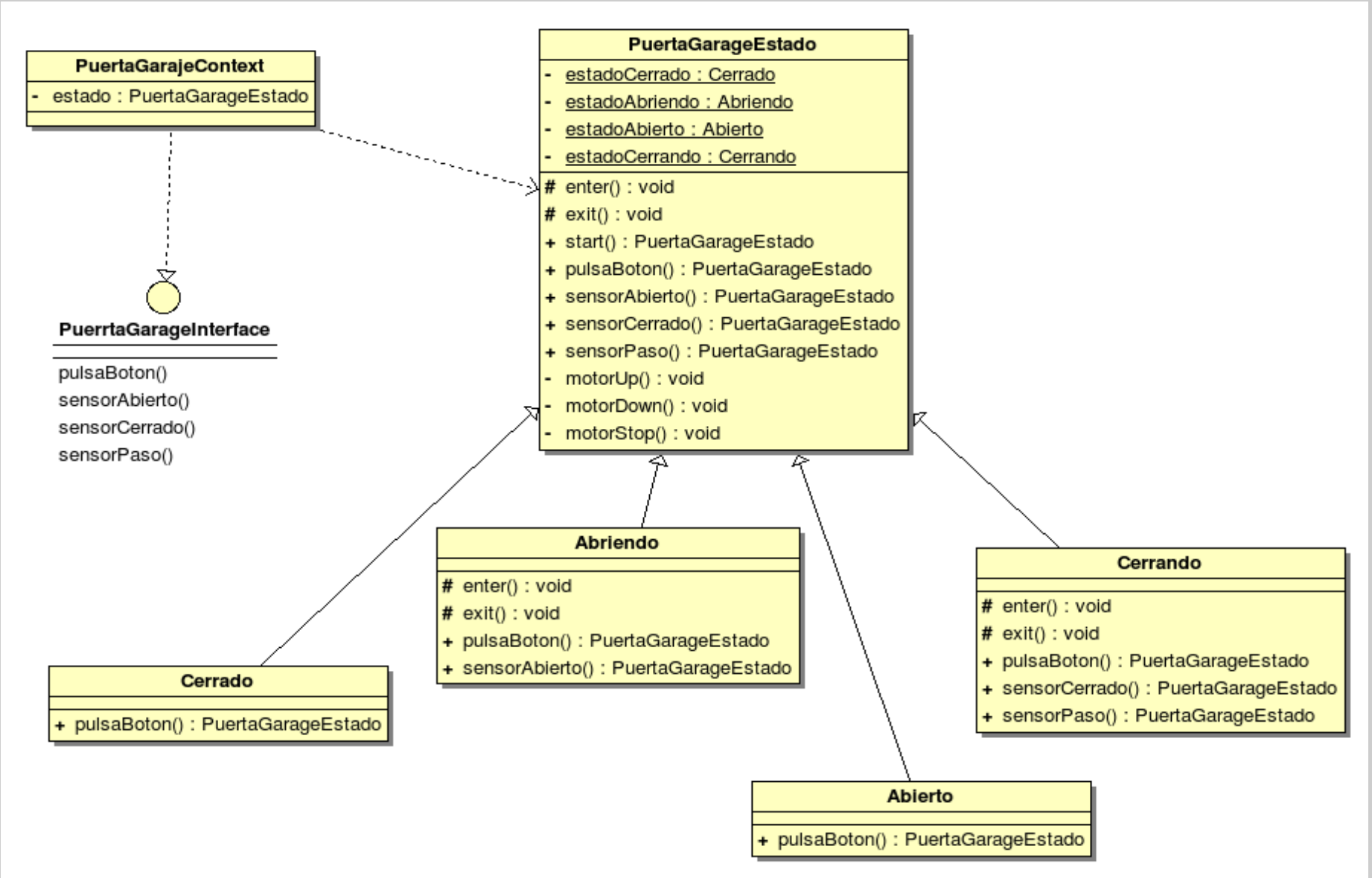
Problemas de la implementación clásica

Dos bloques switch anidados:

- Para el estado actual
- Para el evento

- Añadir un estado significa modificar los 2 bloques switch
  - Y no equivocarse!!.
- El código está articulado en función de los eventos
  - Pero el pensamiento del diseñador está en función de los estados.
- No hay reuso o sobrecarga en función de los estados
  - No se puede reusar acciones comunes entre los diferentes case del switch.
  - No se puede definir atributos específicos a cada estado.

Patrón de máquina de estado



Claves del funcionamiento

La clase `PuertaGarageEstado` fuerza el polimorfismo.

- En realidad la clase contexto (`PuertaGarageContext`) siempre tendrá una referencia a los **objetos de las subclases estado** una vez llama al `arranca()`.
- No puede ser abstracta porque es necesario instanciarla.

- Los abtributos estado (`estadoCerrado` ... `estadoAbriendo`) han de ser **estáticos**.
  - De forma que cuando la clase contexto ejecute los métodos evento (`pulsaBoton()`, etc) los ejecute siempre sobre los mismos objetos (y no atributos de esos objetos).
- La funcionalidad del método `arranca()` no se puede incluir en el constructor de PuertaGarageEstado()
  - De lo contrario aparecerían problemas de recurrencia infinita.
- La clase contexto libera al programa usuario de obligaciones ligadas al patrón.
  - Llamar al método arranca() y recoger el resultado de los métodos evento y sobreescribirlo en la referencia al estado.

Traducción a código (I): Clase Context

```
public class PuertaGarageContexto implements PuertaGarageInterface{

  private PuertaGarageEstadosPattern estado_actual;

  public PuertaGarageContexto()
  {
    estado_actual = new PuertaGarageEstadosPattern();
    estado_actual = estado_actual.arranca();
  }

  public void pulsaBoton()
  {
    estado_actual = estado_actual.pulsaBoton();
  }

  public void sensorAbierto()
  {
    estado_actual = estado_actual.sensorAbierto();
  }

  public void sensorCerrado()
  {
    estado_actual = estado_actual.sensorCerrado();
  }

  public void sensorPaso()
  {
    estado_actual = estado_actual.sensorPaso();
  }
}
```

Traducción a código (II): Clase Estados

```
public class PuertaGarageEstadosPattern {
  private static Cerrado estadoCerrado;
  private static Abriendo estadoAbriendo;
  private static Abierto estadoAbierto;
  private static Cerrando estadoCerrando;

  protected void entry() { }
  protected void exit() { }

  public PuertaGarageEstadosPattern arranca()
  {
    estadoCerrado = new Cerrado();
    estadoAbierto = new Abierto();
    estadoAbriendo = new Abriendo();
    estadoCerrando = new Cerrando();

    estadoCerrado.entry();
    return estadoCerrado;
  }

  // Eventos
  public PuertaGarageEstadosPattern pulsaBoton() { return this; }
  public PuertaGarageEstadosPattern sensorAbierto() { return this; }
  public PuertaGarageEstadosPattern sensorCerrado() { return this; }
  public PuertaGarageEstadosPattern sensorPaso() { return this; }

  // Acciones
  private void motorUp() {
    System.out.println("Motor Up activado");
  }
  private void motorDown() {
    System.out.println("Motor Down activado");
  }
  private void motorStop() {
    System.out.println("Motor Stop activado");
  }

  // SUBCLASSES AQUI
```

Traducción a código (III): Sub-clases estados

```
private class Abriendo extends PuertaGarageEstadosPattern
  {
    protected void entry()
    {
      super.entry();

      motorUp();
    }

    public PuertaGarageEstadosPattern pulsaBoton() {
      this.exit();

      estadoCerrando.entry();
      return estadoCerrando;
    }

    public PuertaGarageEstadosPattern sensorAbierto() {
      this.exit();

      estadoAbierto.entry();
      return estadoAbierto;
    }

    protected void exit()
    {
      motorStop();

      super.exit();
    }


  }
```

Bibliografía

[1] El clásico GoF book:
Gamma, Erich, Richard Helm Ralph Johnson and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

[2] http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29

[3] Mark Grad, *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*, Second Edition. John Wiley & Sons 2002 ISBN 0-471-22729-3

[4] http://sourcemaking.com/design_patterns Patrones del GoF book enseñados para seres humanos.