# Elementos del software orientado a objetos reutilizable

José A. Gallud

# Contenido

El libro Design Patterns

Los lenguajes de programación

Desarrollo ágil de software

Primeros consejos

Claves para comprender los patrones

MVC

Patrones mediante ejemplos

# Sobre el libro Design Patterns

Primera edición: agosto 1994

Título: Design Patterns: Elements of Reusable Object-Oriented Software

Autores: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Es un libro que sirve para medir la madurez y el nivel de nuestro código
Una sola lectura puede resultar insuficiente

# Aha! en vez de Huh?

> *“Once you understand the design patterns and have had an "Aha!" (and not just a "Huh?") experience with them, you won't ever think about object-oriented design in the same way. You'll have insights that can make your own designs more flexible, modular, reusable, and understandable—which is why you're interested in object-oriented technology in the first place, right?” (Design Patterns, Preface to Book)*

No volverás a pensar en diseño orientado a objetos de la misma manera

Los patrones nos llevan a diseñar de manera diferente

# Los lenguajes de programación

Lenguajes OO dinámicos: tipados dinámicamente o sin tipos

Lenguajes que merece la pena aprender:

- Smalltalk: orientado a objetos (con clases)
- Javascript: funcional, orientado a objetos (sin clases)
- Python: orientado a objetos, funcional (con clases)
- Ruby: orientado a objetos, funcional (sin clases)

# Desarrollo ágil de software

Importancia del producto, el equipo y los usuarios

> *Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor*

Desarrollar para el cambio

> *Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo*

Cuatro actividades: codificar, probar, escuchar, diseñar (XP)

# Primeros consejos del buen código

Todo se dice una y sólo una vez

Muchas pequeñas piezas

Tratar de eliminar la lógica condicional

Evitar métodos complejos

Evitar código estructural: un objeto trata a otro como una estructura de datos

Evitar crear super-objetos

# Claves para comprender los patrones

Paradigma Orientado a Objetos:

- Clase: atributos, operaciones
- Objetos, mensajes (peticiones)

Desacoplar

Delegación

Herencia vs composición

Programar para la interface (versus programar para la implementación)

# Un vocabulario enriquecido

Los patrones introducen amplían nuestro vocabulario

Los programadores que conocen los patrones de diseño hablan otro idioma

Patrón:

- Objetivo
- Estructura
- Aplicabilidad
- Consecuencias
- Pistas para implementarlo

# El juego del Laberinto

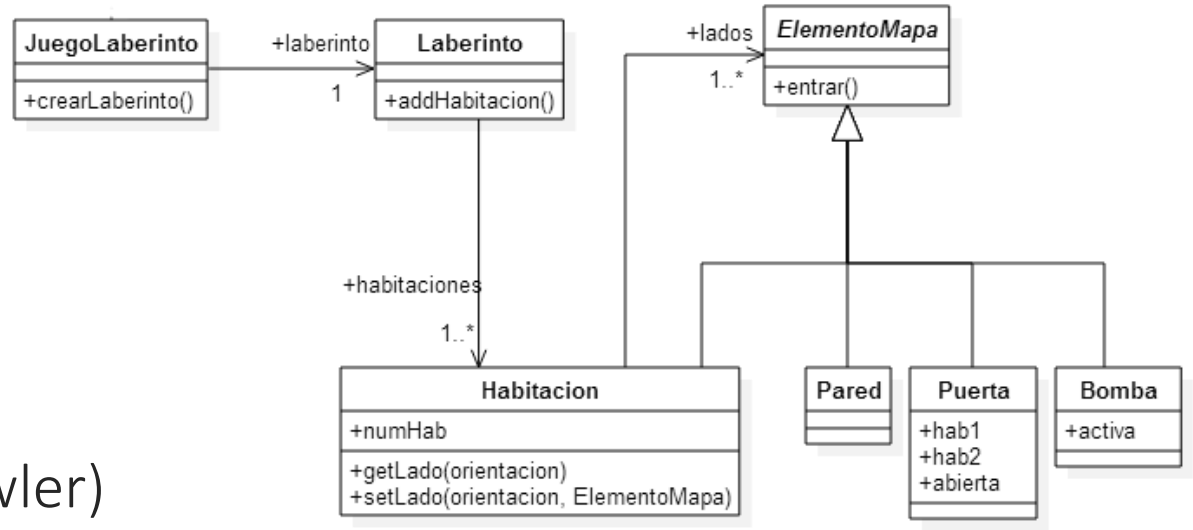
https://github.com/jgallud/patronesJava

Expresiones utilizadas:

- A “tiene un” B
- A “es un” B

Nota:

- agregación vs composición
- asociación vs agregación (Martin Fowler)



# MVC (Modelo-Vista-Controlador)

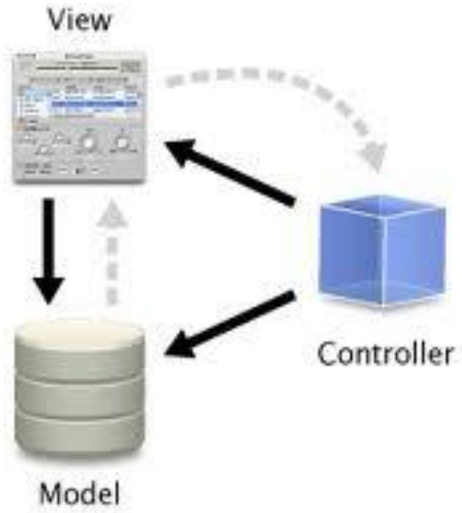
Es un patrón arquitectónico

Se basa principalmente en el patrón Observer

Observer: notificación-suscripción

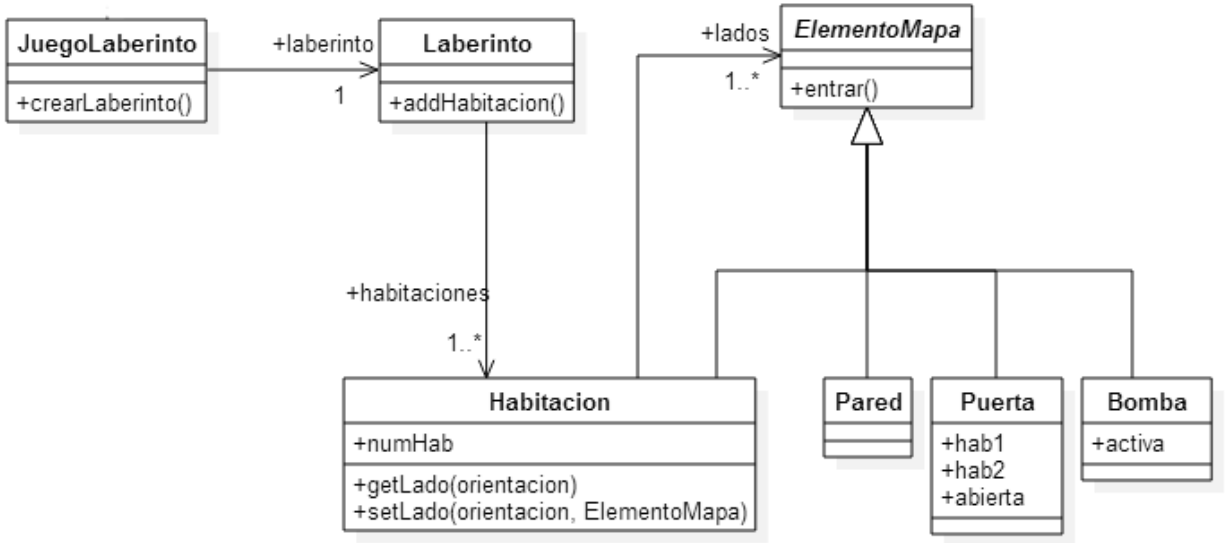
> *Define dependencias uno-a-muchos entre objetos de modo que cuando un objeto cambia su estado, todos sus objetos dependientes reciben una notificación y se actualizan*

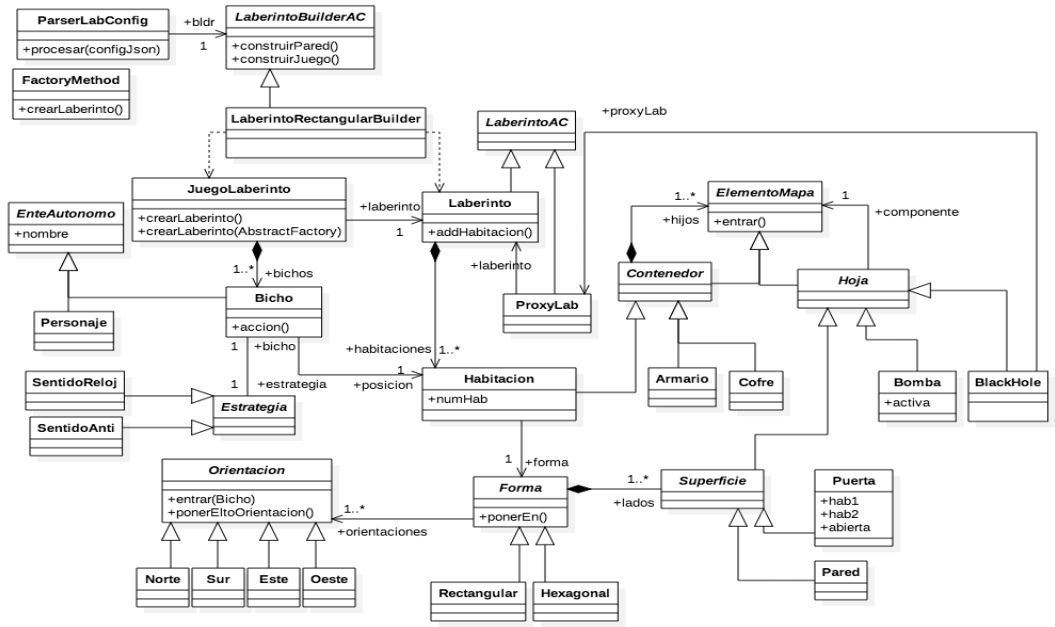
Desacopla las vistas del modelo



# El juego del laberinto

https://github.com/jgallud/patronesJava



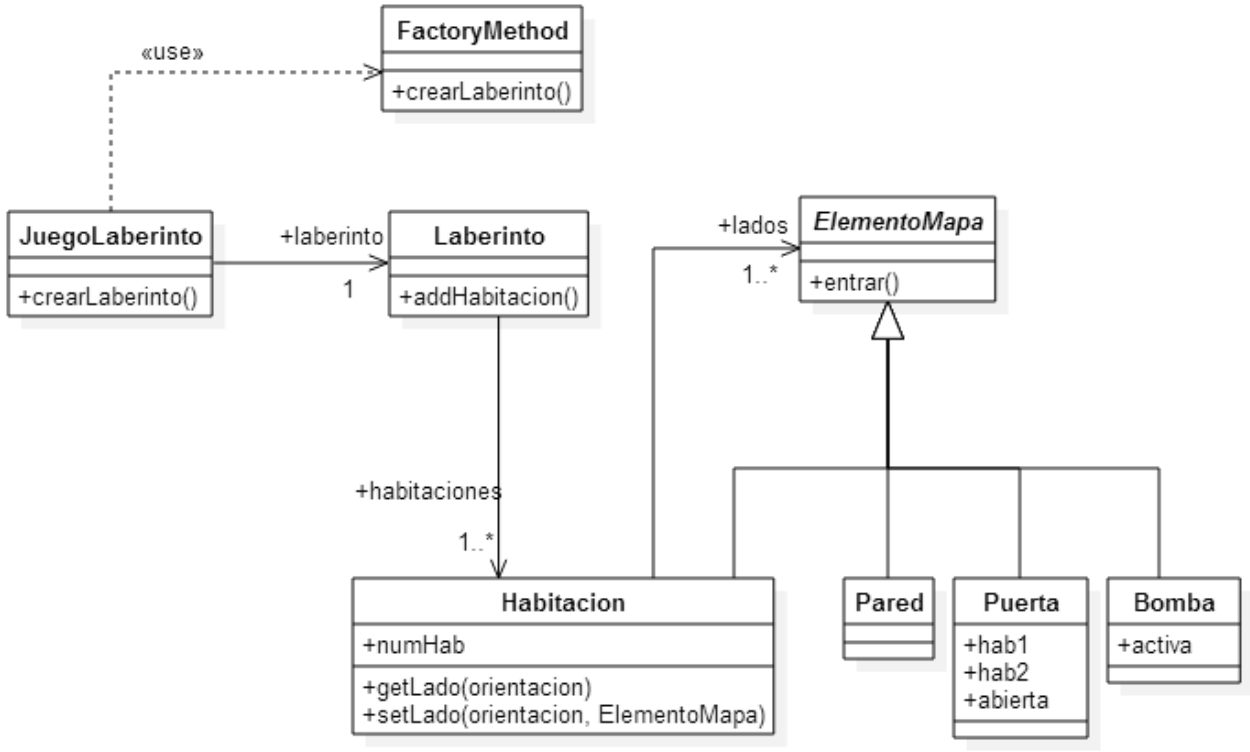



# Factory Method

*Define una interface para crear un objeto pero deja a las subclases decidir qué objeto crear*

Puedo tener new Laberinto() en JuegoLaberinto, o bien:

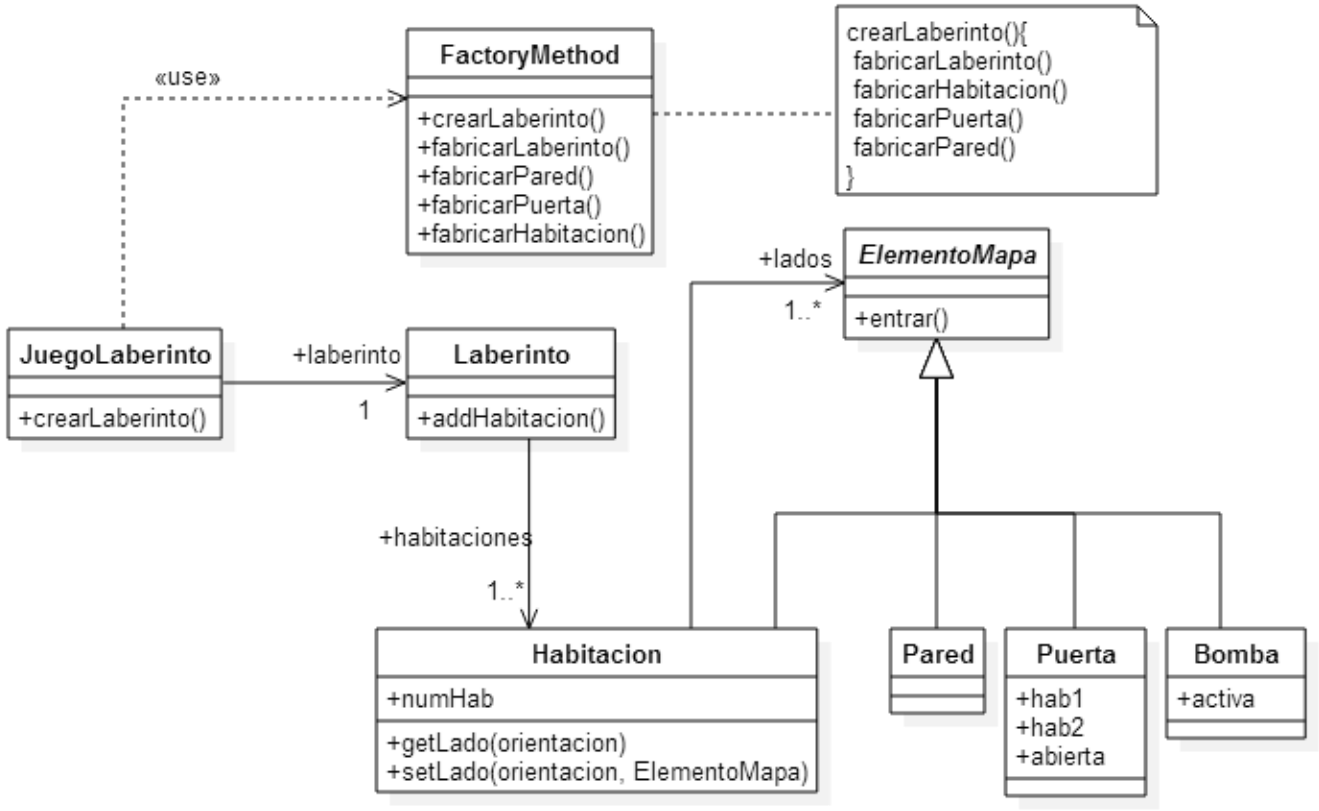
laberinto=fm.crearLaberinto()



# Template Method

*Define el esqueleto de un algoritmo en una operación y deja algunos pasos a las subclases*

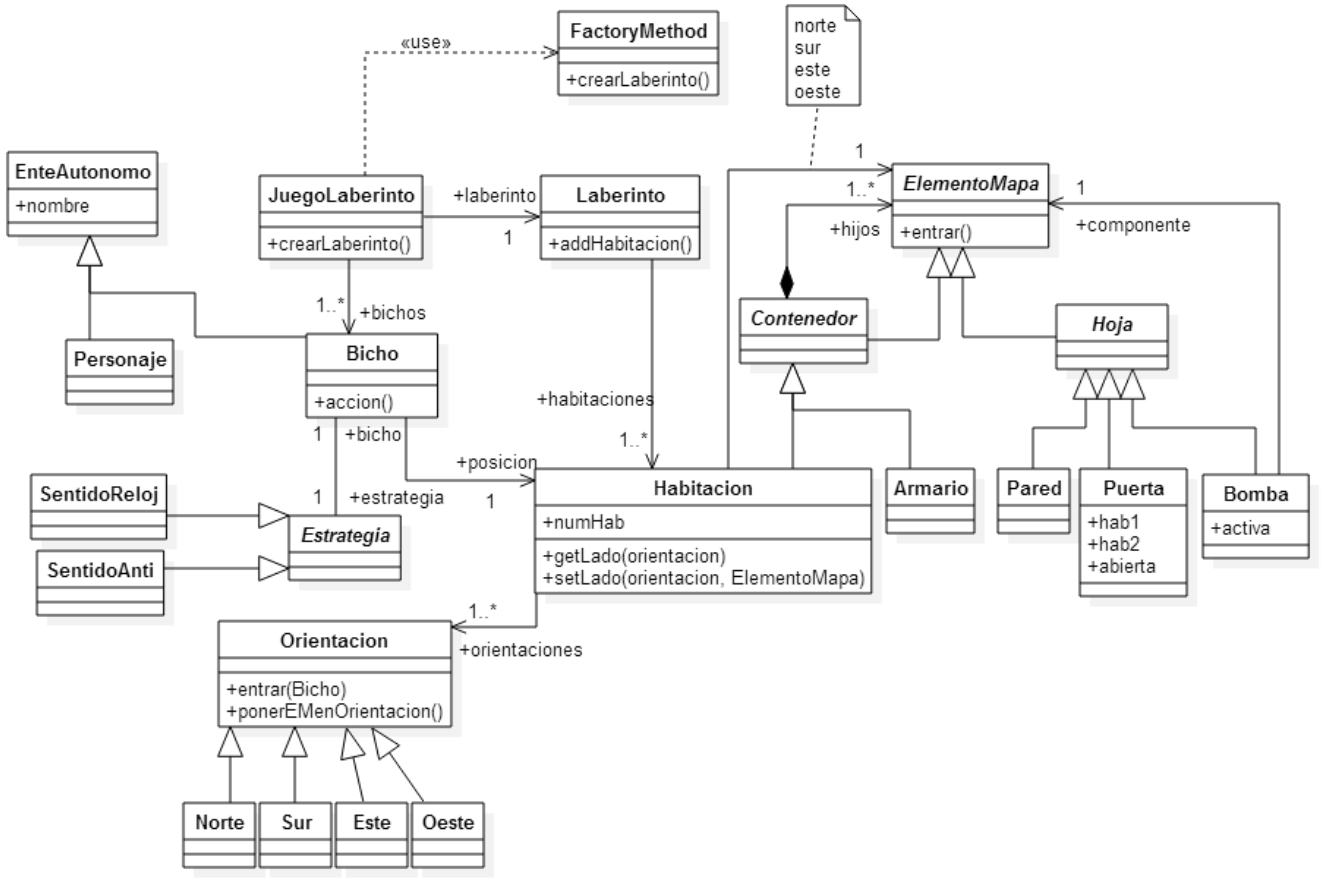
Podemos crear subclases de FactoryMethod que nos permitan crear otro tipo de laberintos



# Composite

*Compone objetos en una estructura en árbol para representar jerarquías todo-parte*

El cliente accede de forma uniforme a objetos de tipo contenedor y de tipo hoja

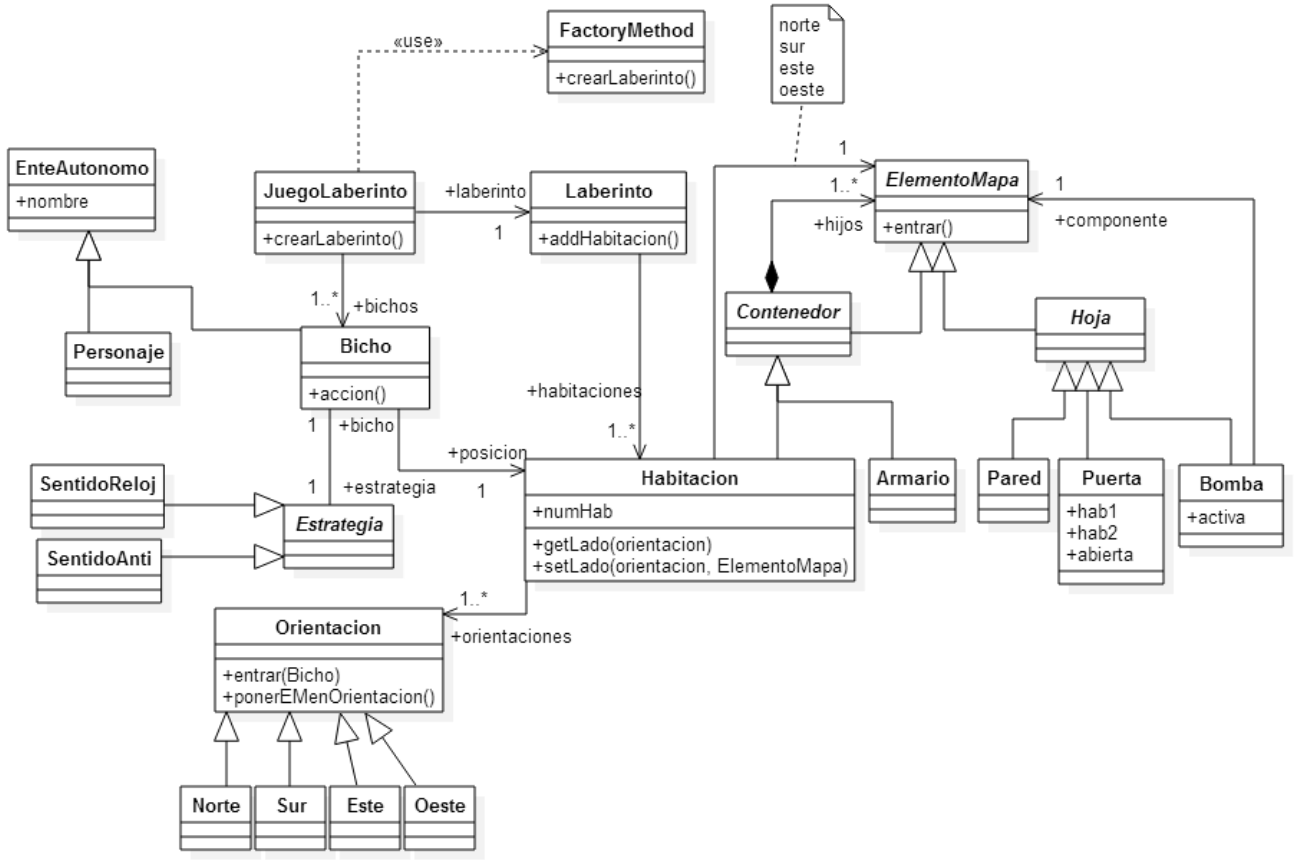


# Decorator

*Agrega responsabilidades adicionales a un objeto de forma dinámica*

Alternativa al Decorator:

- ParedBomba
- PuertaBomba
- HabitacionBomba



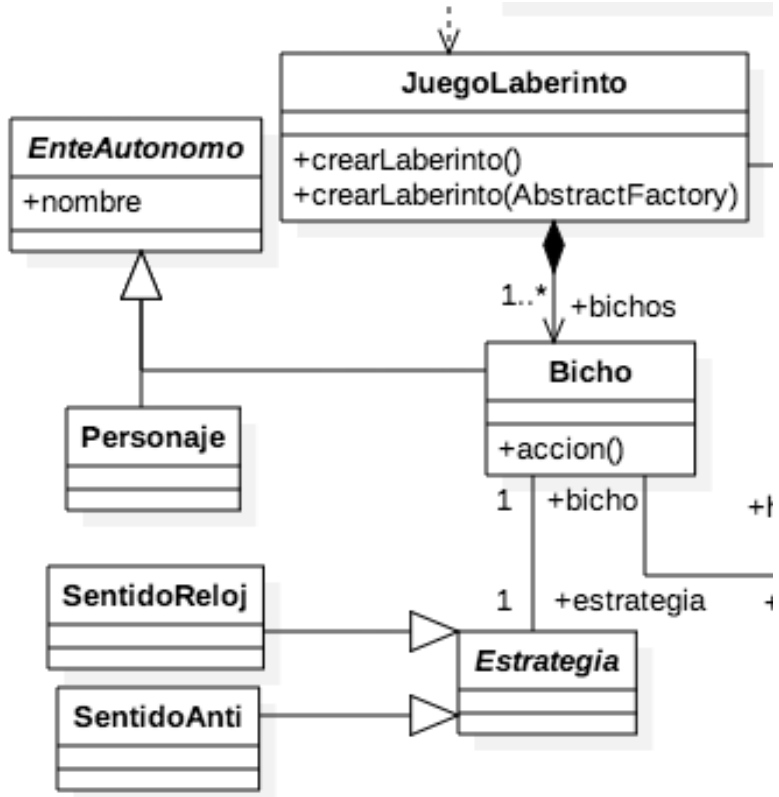
# Strategy

*Define una familia de algoritmos y encapsula cada uno en un objeto de modo que se pueden intercambiar*

Es una alternativa a la herencia:

- BichoSentidoReloj
- BichoSentidoAnti

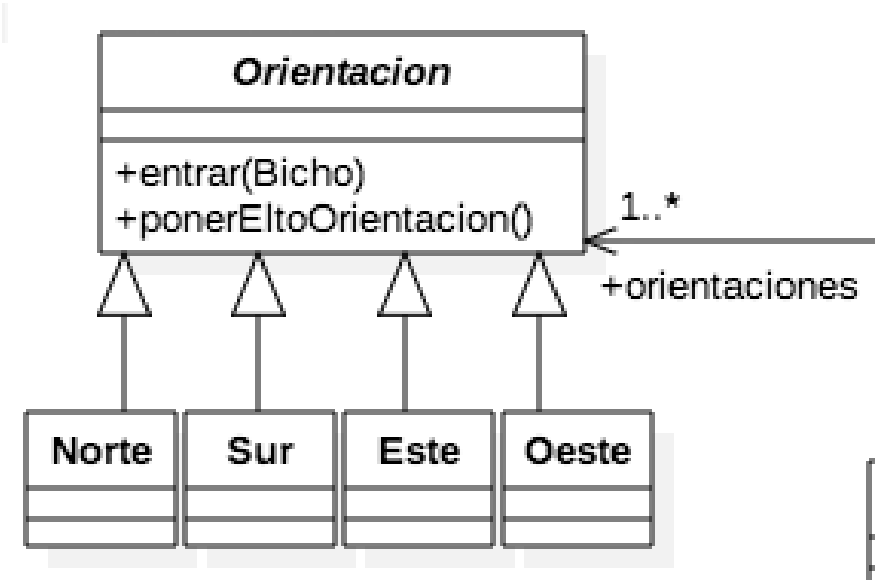
Utiliza composición



# Strategy (2)

```
protagonista.irHacia(unaOrientacion)
//Supongamos la siguiente implementación
switch(unaOrientacion){
        case "norte":…;break;
        case "este":…;break;
        }

//alternativa utilizando delegación:
public void irHacia(Orientacion or){
                or.irHacia(this);
        }
```

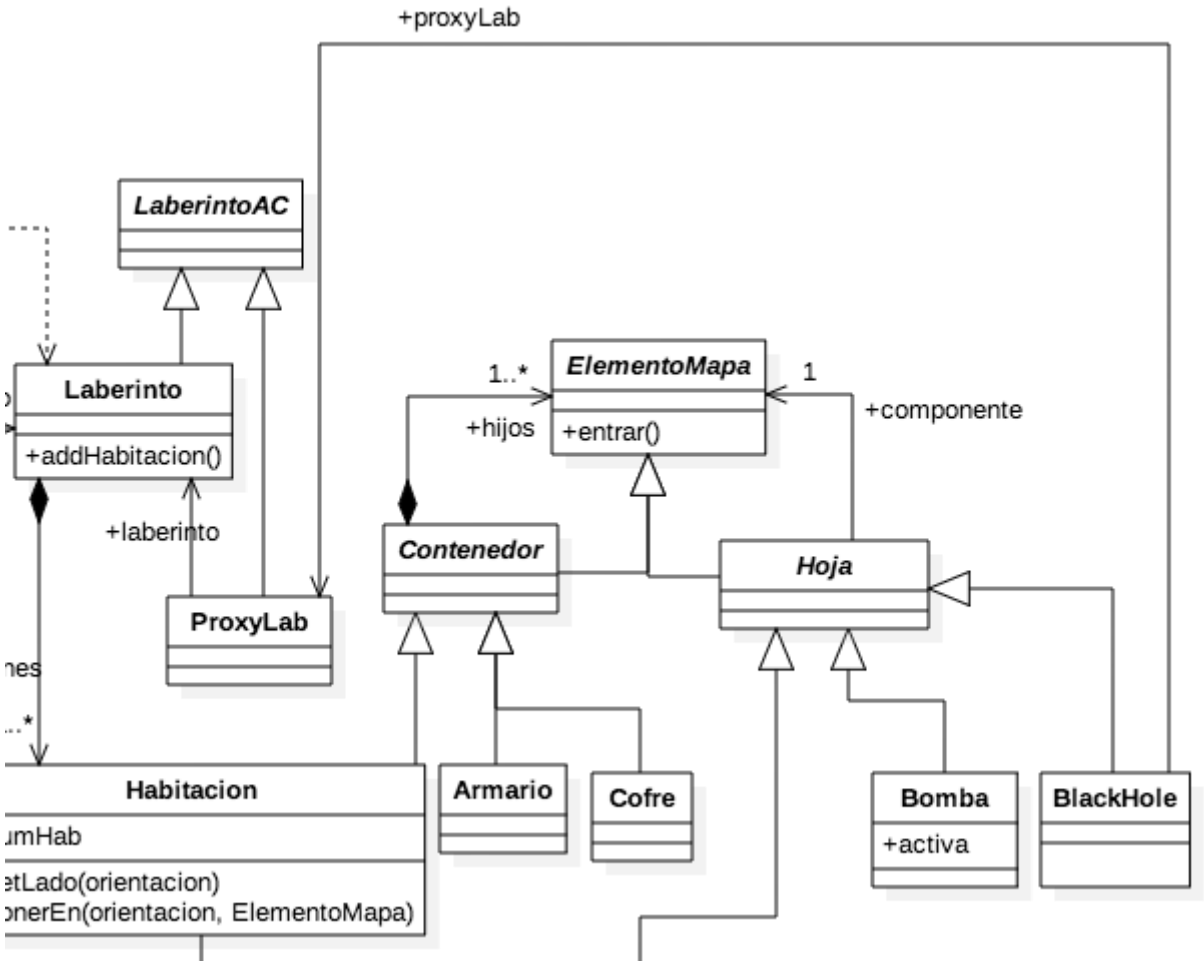


# Proxy

*Proporciona un representante de otro objeto para controlar el acceso a éste*

Necesitamos un elemento nuevo que al entrar en él, el usuario se meta en un nuevo laberinto

BlackHole tiene un laberinto (ProxyLab) que se crea sólo cuando el usuario entra en el BlackHole



# Command

*Encapsula una petición como un objeto*

Permite parametrizar los clientes con diferentes peticiones

# State

*Permite a un objeto alterar su comportamiento cuando cambia su estado interno*

Cada estado de un objeto es, a su vez, otro objeto

# Otros patrones

| | |
|---|---|
| Bridge | Separa abstracción de implementación |
| Chain of Responsibility | Desacopla emisor y receptor de una petición |
| Memento | Almacena el estado de un objeto en otro objeto |
| Iterator | Permite recorrer un objeto de forma secuencia |
| Visitor | Define una nueva operación en objeto compuesto |
| Abstract Factory | Permite crear familias de productos |
| Builder | Separa la construcción de un objeto de su representación |
| Singleton | Asegura que sólo hay una instancia de un tipo de objeto |
| Mediator | Encapsula en un objeto las interacciones de un grupo de objetos |

# Aplicación práctica

Javascript:

- No hay clases → no uso herencia