Tecnología de Programación

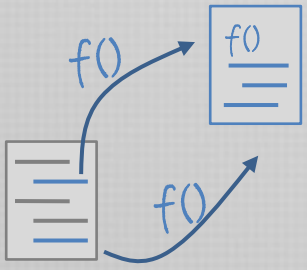
Martín L. Larrea

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

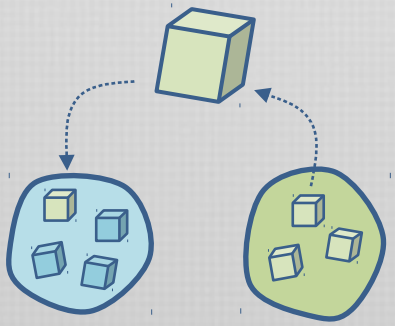
Reutilización

La reutilización es un ingrediente fundamental en la ingeniería de software

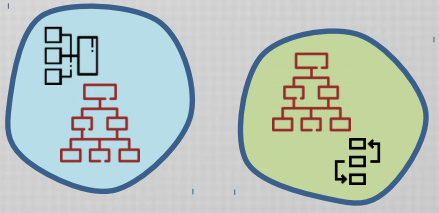
Ofrece varios beneficios:
confiabilidad, menor tiempo de desarrollo, menos costos



Reutilización
de código



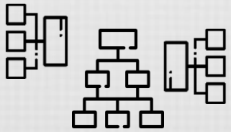
Reutilización
de módulos



Reutilización
de diseño

Varios mecanismos técnicos facilitan la reutilización:
clases, herencia, composición de objetos, genericidad

Reutilización



En general, muchas de las dificultades en alcanzar
un buen programa radican en el diseño.
El diseño es realmente la forma, la arquitectura del sistema.
El lenguaje de programación es sólo una herramienta de implementación

En orientación a objetos, el buen diseño requiere un buen estudio...

Siempre pensando en la reutilización del software, debemos:

encontrar los objetos pertinentes,
factorizarlos en clases con una granularidad adecuada,
definir las interfaces,
establecer relaciones entre clases...

El diseño debe ser adecuado para la aplicación en particular
El diseño debe ser lo mas general y flexible posible.

Reutilizar diseño

Los programadores y desarrolladores experimentados
no resuelven las cosas desde cero.
Reutilizan soluciones previamente encontradas, testeadas y aprobadas.

Lo hacen en el código con facilidad al identificar
el procedimiento a implementar.

Lo hacen en el diseño con facilidad al identificar la aplicación particular.

Básicamente, identifican patrones y aplican soluciones.

Nos centraremos principalmente en la etapa de diseño (tal vez la más difícil) y analizaremos diferentes soluciones que pueden servirnos
en el futuro para desarrollar buenas aplicaciones.

Esa es la idea general de los patrones de diseño.

Patrones de diseño

Los patrones de diseño nombran, explican y evalúan un diseño importante y recurrente en los sistemas orientados a objetos.

En general, poseen estos cuatro elementos:

Nombre del patrón
para poder identificarlo fácilmente y tratarlo
de manera abstracta cuando sea necesario.

El problema
que describe cuándo utilizar este patrón de forma tal que sea la solución.

La solución
que describe los elementos que componen el diseño, sus relaciones, sus responsabilidades, etc. Se describe en forma abstracta, pues un patrón es un template que se aplica en diferentes situaciones.

Las consecuencias,
que son los resultados y el balance final de aplicar el patrón
(impacto en el sistema, detalles de lenguajes,etc)

Patrones de Diseño, según GoF

Los patrones de diseño son básicamente descripciones de objetos que se comunican y clases que son personalizadas para resolver un problema de diseño general en un contexto particular [GoF]

Gang
Of
Four


Erich Gamma


Ralph Johnson


John Vlissides


Richard Helm

Los patrones se describen gráficamente,
lo que facilita su comprensión, pero no es suficiente.
Algunos aspectos no pueden ser aclarados o especificados por medio de diagramas.

Se adopta entonces una convención para la descripción de patrones.

Es un formato generalmente aceptado que incluye todos los *items* a destacar.

Descripción de patrones de diseño

- Nombre del patrón y clasificación
- Intención: ¿qué hace? ¿qué problema ataca?
- Alias: algunos patrones son conocidos por nombres diferentes.
- Motivación: un escenario que describe el problema de diseño y que muestra como el patrón resuelve el problema.
- Aplicabilidad: cuáles son las situaciones en las cuales se aplica el patrón.
- Estructura: representación gráfica de las clases del patrón (diagramas de clases, diagramas de interacción)
- Participantes: clases y objetos que participan del patrón.
- Colaboraciones: cómo los participantes colaboran para realizar alguna tarea.
- Consecuencias: beneficios, balance pros y contras, etc.
- Implementación: hints, técnicas y detalles a tener en cuenta al implementar el patrón
- Código ejemplo: fragmentos de código que ejemplifican la implementación
- Usos conocidos: ejemplos de patrones en sistemas reales.
- Patrones relacionados: qué otros patrones de diseño están relacionados o pueden combinarse para resolver problemas mayores.

Los siguientes son los patrones de diseño conocidos como GoF

		PROPÓSITO		
		CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
SCOPE	CLASE	Factory Method	Adapter	Interpreter Template Method
	OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Patrones Creacionales

Creational Patterns

Patrones creacionales

Los *patrones creacionales* son patrones que abstraen el proceso de instanciación.

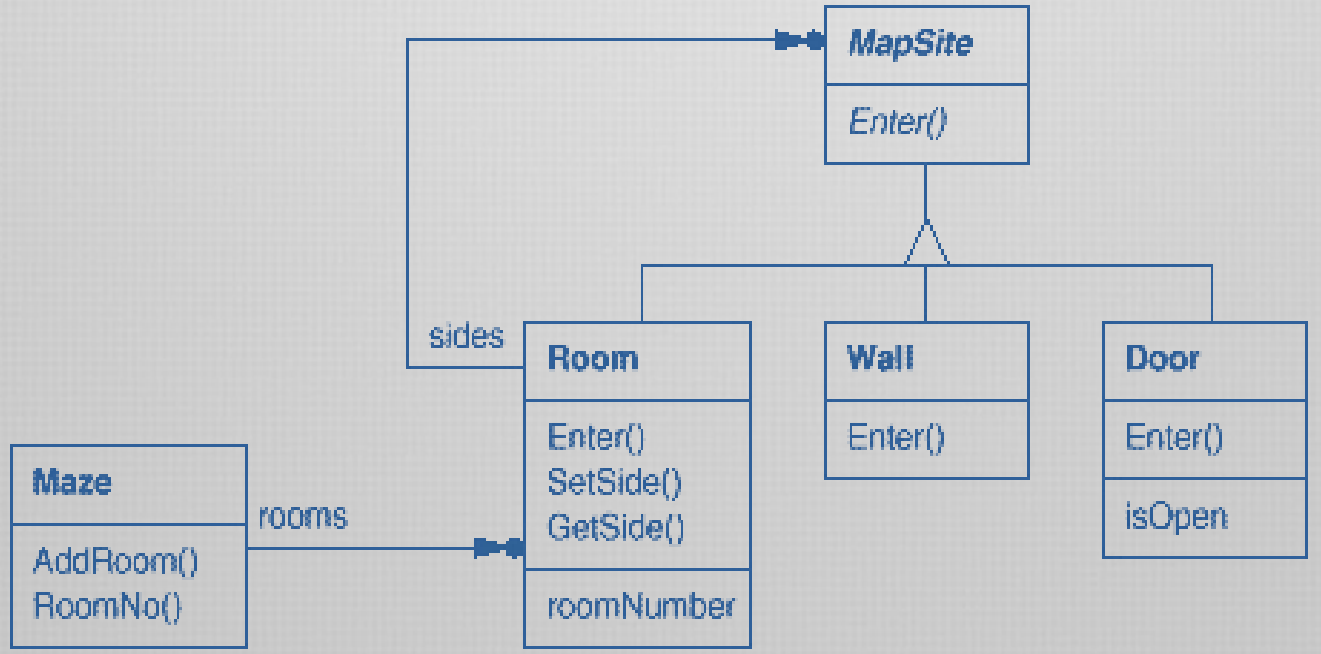
Procuran independizar el sistema de cómo sus objetos son creados, compuestos y representados.

Los patrones indican soluciones de diseño para encapsular el conocimiento acerca de las clases que el sistema usa y ocultar cómo se crean instancias de estas clases.

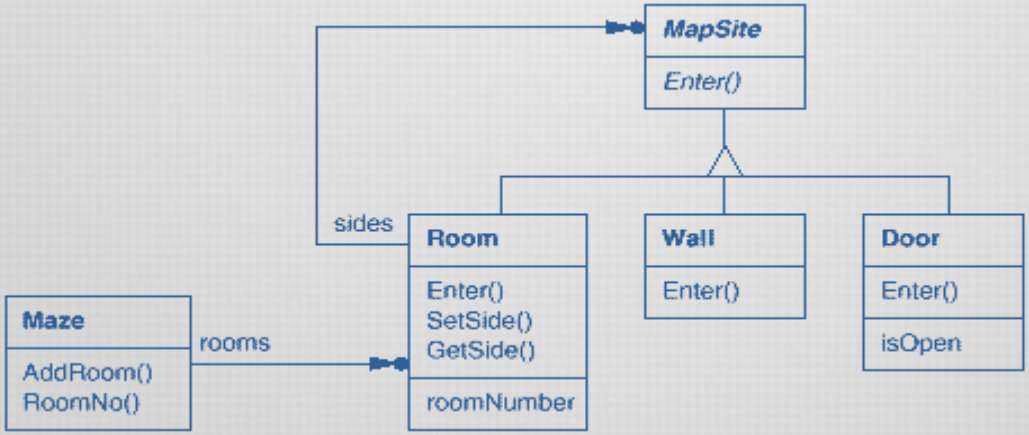
Seguiremos el ejemplo que *Gamma et al.* proponen para estudiar estos patrones: la creación de un laberinto para juegos.

Patrones creacionales - ejemplo

Un laberinto (maze) es un conjunto de habitaciones (rooms). Una habitación conoce a sus vecinos, los cuales pueden ser otra habitación, un muro o una puerta a otra habitación.



Patrones creacionales - ejemplo



Cada habitación tiene cuatro lados. En C++ podemos declarar un enumerado:

```
enum Direction {North, South, East, West};
```

La clase *MapSite* es una clase abstracta para todos los componentes del laberinto. Posee una sola operación *Enter()* para simplificar.

El significado de *Enter()* depende del componente.
Si es una habitación, cambiamos de locación,
si es una puerta y está abierta, pasamos a la siguiente habitación.

Patrones creacionales – ejemplo - C++

```
class Room : public MapSite {
public:
    Room(int roomNo);
    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);
    virtual void Enter();
private:
    MapSite* _sides[4];
    int _roomNumber;
};
```

```
class Wall:public MapSite
{
public:
  Wall();
  virtual void Enter();
};
```

```
class Door : public MapSite {
public:
  Door(Room* = 0, Room* = 0);
  virtual void Enter();
  Room* OtherSideFrom(Room*);
private:
  Room* _room1; Room* _room2;
  bool _isOpen;
};
```

```
class Maze {
public:
    Maze();
    void AddRoom(Room*);
    Room* RoomNo(int)const;
private:
    // ...
};
```

Patrones creacionales – ejemplo - C++

En la clase MazeGame necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);
    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);
    return aMaze;
}
```

Esta operación crea un laberinto con dos habitaciones

Puede simplificarse. Por ejemplo, las habitaciones podrían crear las paredes.

Pero esto sólo mueve código de un lugar a otro

Además ¿Qué pasa si queremos agregar otro elemento al laberinto o modificar uno existente?

Los patrones creacionales procuran hacer este diseño más flexible, quitando las referencias explícitas a clases concretas.

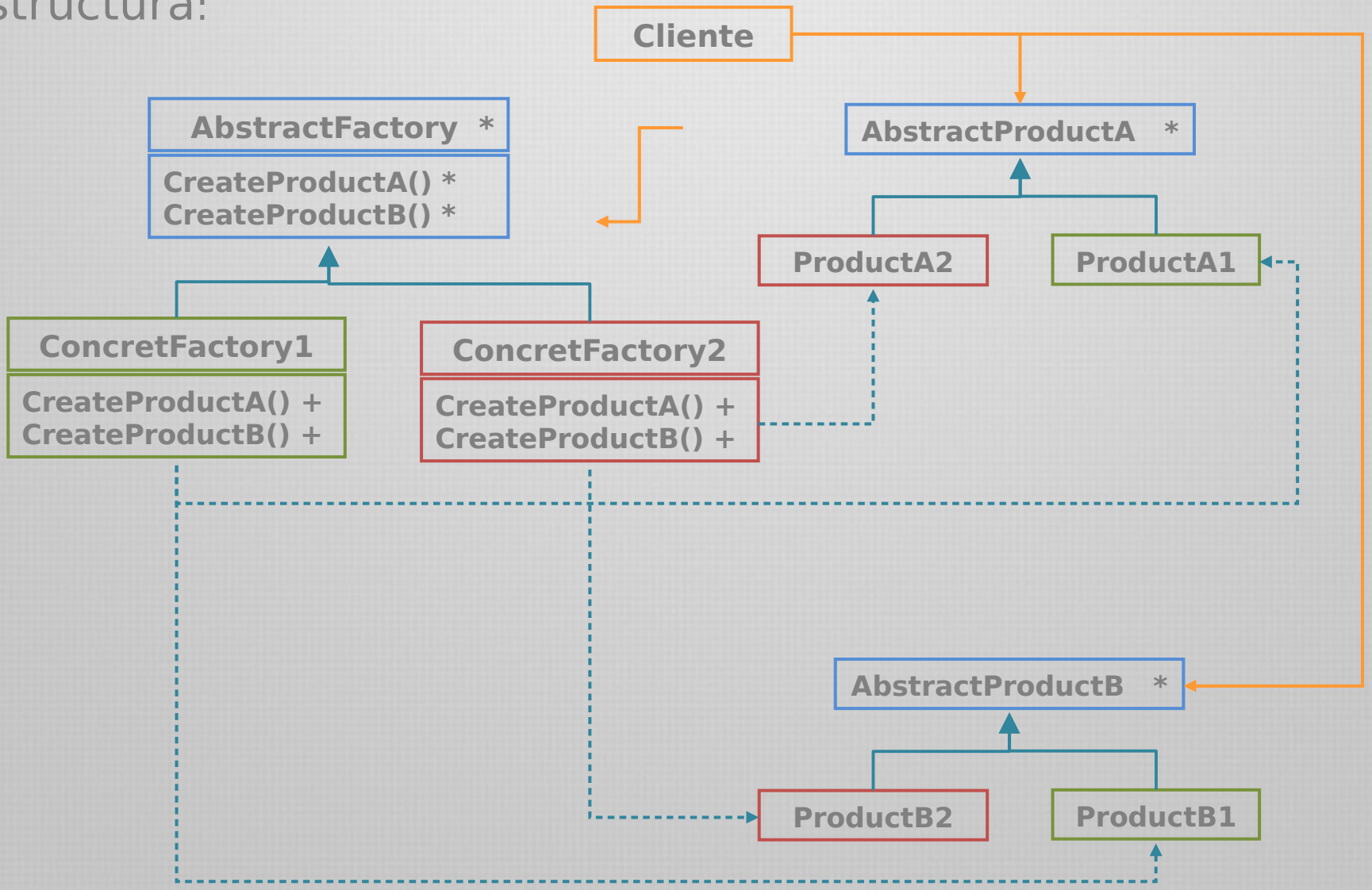
Patrón: Abstract Factory

- Intención: Proveer una interfaz para crear familias de objetos dependientes o relacionados sin especificar sus clases concretas.
- Alias: Kit (como sufijo usualmente).
- Aplicabilidad: Usamos este patrón cuando
  - Un sistema debe independizarse de cómo sus productos son creados, compuestos y representados.
  - Un sistema debe ser configurado con una de múltiples familias de productos.
  - Una familia de objetos debe ser usada en conjunto, y debe reforzarse este hecho.
  - Queremos proveer una librería de productos, pero sólo publicitar las interfaces, no las implementaciones.

Este patrón nos dice cómo definir una fábrica de objetos relacionados

Patrón: Abstract Factory

Estructura:



Patrón: Abstract Factory

Participantes:

- AbstractFactory: declara una interfaz para operaciones que crean objetos producto (abstractos)
- ConcreteFactory: implementa las operaciones para crear objetos producto concretos
- AbstractProduct: declara una interfaz para un tipo de producto
- ConcreteProduct: define un objeto producto que será creado por la correspondiente clase factory concreta. Implementa la interfaz *AbstractProduct*.
- Cliente: usa sólo interfaces declaradas por *AbstractFactory* y *AbstractProduct*.

Patrón: Abstract Factory en laberintos

Apliquemos el patrón *Abstract Factory* a los laberintos...

```
class MazeFactory {
   public:
    MazeFactory();
    virtual Maze* MakeMaze() const
       { return new Maze; }
    virtual Wall* MakeWall() const
       { return new Wall; }
    virtual Room* MakeRoom(int n) const
       { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
       { return new Door(r1, r2); }
};
```

La clase *MazeFactory* crea componentes clásicos de laberinto.

Patrón: Abstract Factory en laberintos

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
   Maze* aMaze = factory.MakeMaze();
   Room* r1 = factory.MakeRoom(1);
   Room* r2 = factory.MakeRoom(2);
   Door* aDoor = factory.MakeDoor(r1, r2);
   aMaze->AddRoom(r1);
   aMaze->AddRoom(r2);
   r1->SetSide(North, factory.MakeWall());
   r1->SetSide(East, aDoor);
   r1->SetSide(South, factory.MakeWall());
   r1->SetSide(West, factory.MakeWall());
   r2->SetSide(North, factory.MakeWall());
   r2->SetSide(East, factory.MakeWall());
   r2->SetSide(South, factory.MakeWall());
   r2->SetSide(West, aDoor);

   return aMaze;
}
```

Patrón: Abstract Factory en laberintos

Podemos crear otros tipos de laberintos simplemente utilizando herencia e invocando la operación anterior con el *factory* correspondiente.

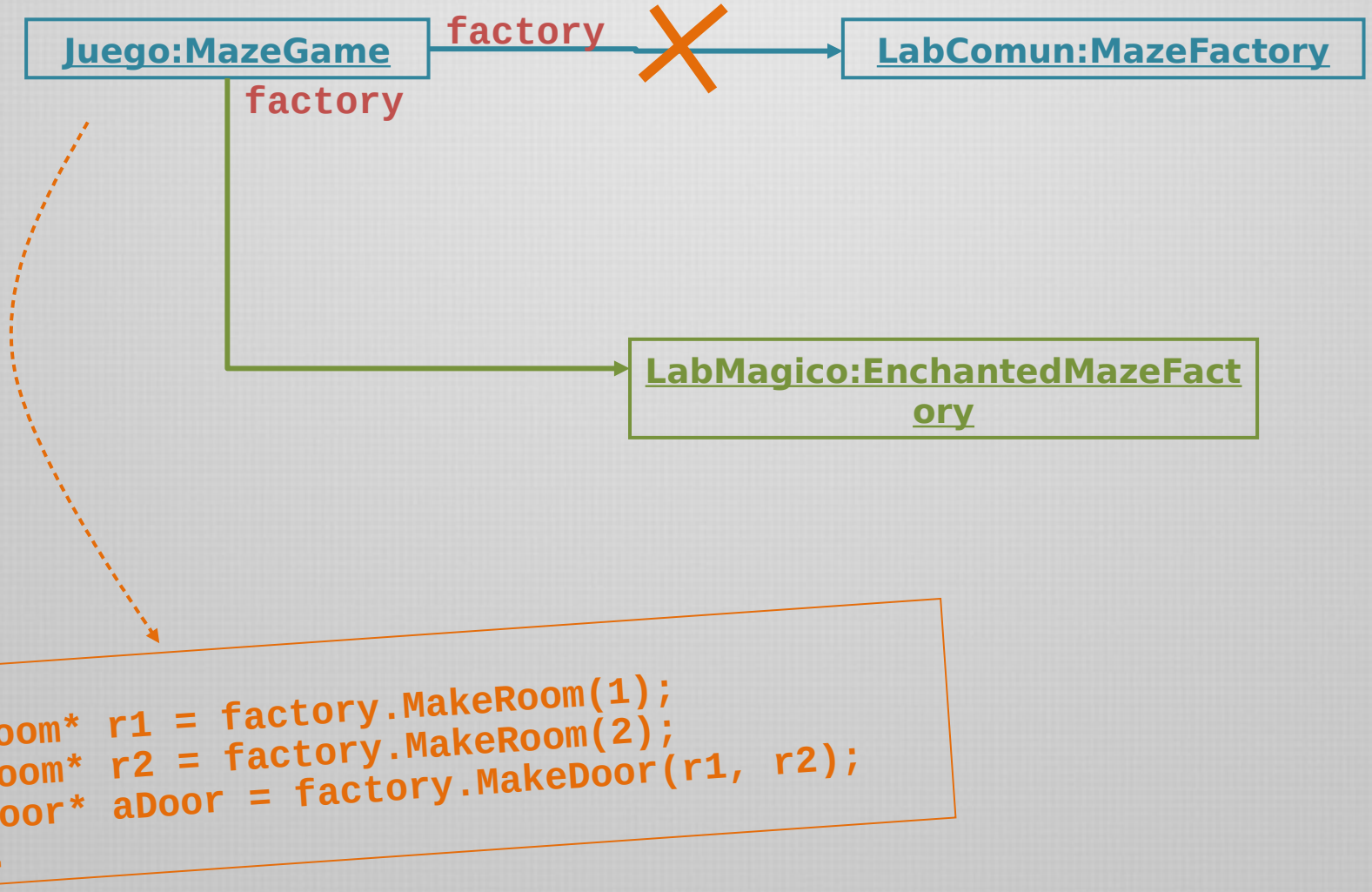
```
class EnchantedMazeFactory : public MazeFactory {
   public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n) const {
           return new EnchantedRoom(n, CastSpell());
    }

    virtual Door* MakeDoor(Room* r1, Room* r2) const {
           return new DoorNeedingSpell(r1, r2);
    }

   protected:
    Spell* CastSpell() const;
};
```

Patrón: Abstract Factory en laberintos




1P
HIGH SCORE
24840
10328300
00


PLAYER 1
HIGH SCORE
PLAYER 2
Game over