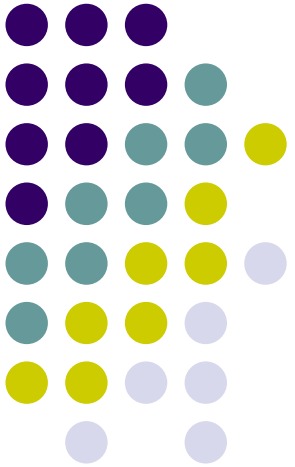
Patrones de Diseño

Patrones de creación



Patrones de creación

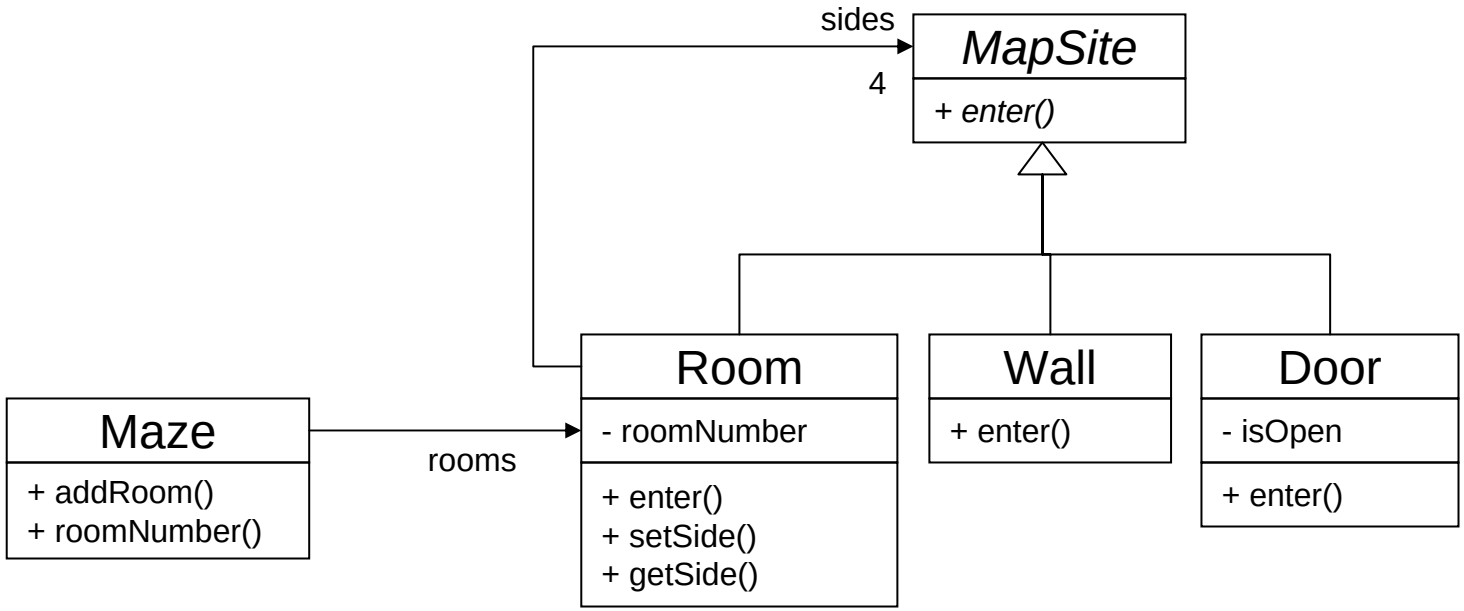
Introducción

- Abstraen el proceso de instanciación
- Encapsulan conocimiento sobre qué clases concretas utiliza el sistema
- Independizan al sistema del modo en que se crean, componen y representan los objetos
- Flexibilizan el qué, quién, cómo y cuándo

  - *Factory method* (patrón de creación de clase)
  - *Abstract factory* (patrón de creación de objeto)
  - *Singleton* (patrón de creación de objeto)

Patrones de creación

Ejemplo: laberinto



Patrones de creación

Ejemplo: laberinto


```
public enum Direction {
   NORTH,
   SOUTH,
   EAST,
   WEST
}

public abstract class MapSite {
   abstract void enter();
}

public class Room extends MapSite {
   private int _roomNumber;
   private MapSite _sides[]=new MapSite[4];
   Room () {}
   Room (int n) { _roomNumber = n; }
   MapSite getSide (Direction dir) {
      return _sides[dir.ordinal()];
   }
   void setSide (Direction dir, MapSite s){}
   void enter() {}
}
```

```
public class Wall extends MapSite {
   Wall () {}
   void enter() {}
}

public class Door extends MapSite {
   private Room _room1;
   private Room _room2;
   private boolean _isOpen;
   Door (Room r1, Room r2) {}
   void enter() {}
   Room otherSideFrom (Room r1) {}
}

public class Maze {
   Maze() {}
   void addRoom (Room r) {}
   Room RoomNumber (int n) { }
}
```

Patrones de creación

Ejemplo: laberinto


```
public class MazeGame {
   Maze createMaze () {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door aDoor = new Door(r1, r2);

      aMaze.addRoom(r1);
      aMaze.addRoom(r2);

      r1.setSide(Direction.NORTH, new Wall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, new Wall());
      r1.setSide(Direction.WEST,  new Wall());
      r2.setSide(Direction.NORTH, new Wall());
      r2.setSide(Direction.EAST,  new Wall());
      r2.setSide(Direction.SOUTH, new Wall());
      r2.setSide(Direction.WEST,  aDoor);

      return aMaze;
   }
}
```

Patrones de creación
Ejemplo: laberinto


```
public class MazeGame {
   Maze createMaze () {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door aDoor = new Door(r1, r2);

      aMaze.addRoom(r1);
      aMaze.addRoom(r2);

      r1.setSide(Direction.NORTH, new Wall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, new Wall());
      r1.setSide(Direction.WEST,  new Wall());
      r2.setSide(Direction.NORTH, new Wall());
      r2.setSide(Direction.EAST,  new Wall());
      r2.setSide(Direction.SOUTH, new Wall());
      r2.setSide(Direction.WEST,  aDoor);

      return aMaze;
   }
}
```

Largo: cuatro llamadas a `setSide` por habitación. Podemos inicializar la habitación en el constructor

Poco flexible:

- otras formas de laberinto?
  - cambiar método
  - añadir nuevo método
- otros tipos de laberinto?

Patrones de creación
Ejemplo: laberinto


```
public class MazeGame {
   Maze createMaze () {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door aDoor = new Door(r1, r2);

      aMaze.addRoom(r1);
      aMaze.addRoom(r2);

      r1.setSide(Direction.NORTH, new Wall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, new Wall());
      r1.setSide(Direction.WEST,  new Wall());
      r2.setSide(Direction.NORTH, new Wall());
      r2.setSide(Direction.EAST,  new Wall());
      r2.setSide(Direction.SOUTH, new Wall());
      r2.setSide(Direction.WEST,  aDoor);

      return aMaze;
   }
}
```

Factory method: funciones de creación en vez de constructores => cambiar el tipo de lo que se crea mediante redefinición

Abstract factory: objeto para crear los objetos => cambiar el tipo de lo que se crea recibiendo un objeto distinto

Singleton: un único objeto laberinto en el juego

Factory method

Propósito

- Define una interfaz para crear un objeto, pero dejando en manos de las subclases la decisión de qué clase concreta instanciar
- Permite que una clase delegue en sus subclases la creación de objetos
- También conocido como *virtual constructor*

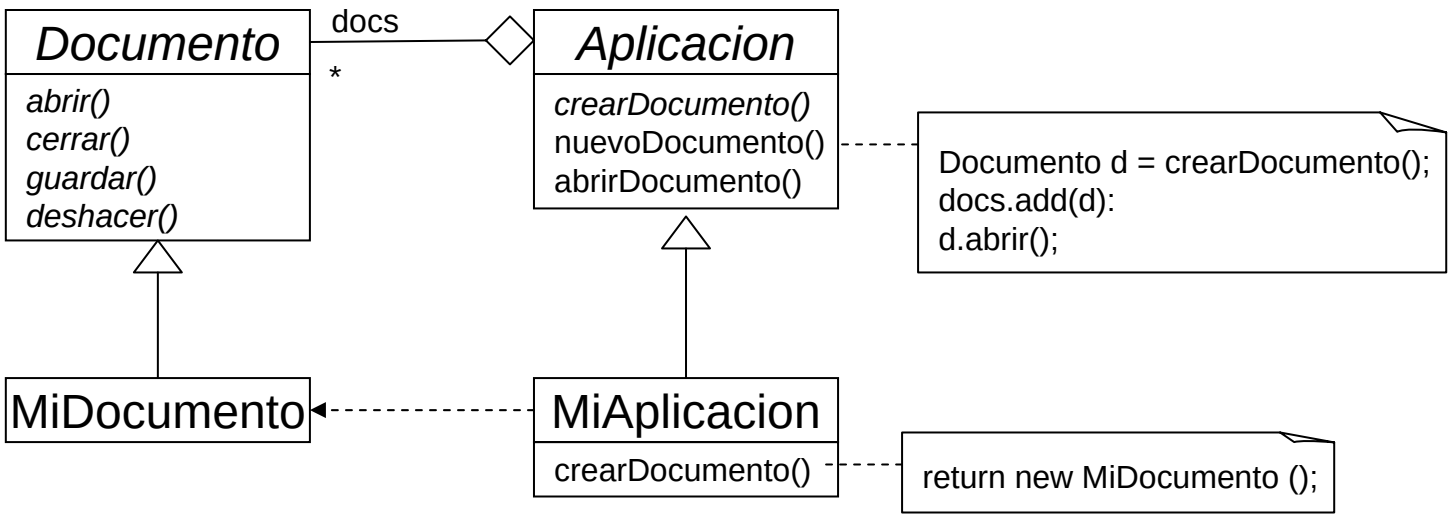
Factory method
Motivación

- Ej: un framework de aplicaciones debe poder presentar distintos tipos de documentos

- El framework maneja dos abstracciones:
  - *Documento*: los distintos tipos se definen como subclases
  - *Aplicacion*: sabe cuándo crear un documento, pero no su tipo (no puede predecir el tipo de documento que el programador definirá).

- Solución:
  - Encapsular el conocimiento sobre qué subclase de *Documento* crear, y mover ese conocimiento fuera del framework

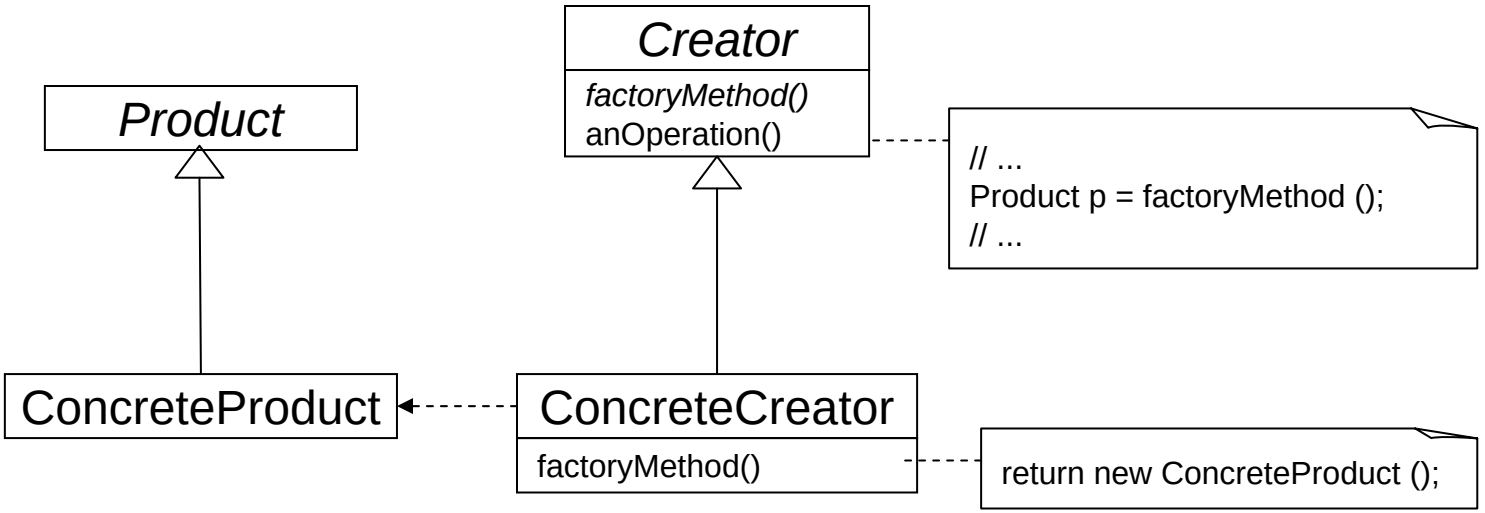
Factory method
Motivación



Factory method
Aplicabilidad

- Usa el patrón *Factory method* cuando:
  - Una clase no puede prever la clase de objetos que tiene que crear
  - Una clase quiere que sus subclases decidan qué objetos crean
  - Las clases delegan responsabilidades a una de entre varias subclases auxiliares, y queremos localizar en qué subclase concreta se ha delegado

Factory method
Estructura



Factory method
Participantes

- **Product** (*Documento*): define la interfaz de los objetos que crea el método factoría
- **ConcreteProduct** (*MiDocumento*): implementa la interfaz de *Product*
- **Creator** (*Aplicacion*)*:*
  - declara el método factoría que devuelve un objeto de tipo *Product*. Puede definir una implementación por defecto de dicho método, que devuelva un objeto de algún producto concreto *ConcreteProduct*.
  - puede llamar al método factoría para crear un objeto de tipo *Product*
- **ConcreteCreator** (*MiAplicacion*)*:* sobrescribe el método factoría para devolver un objeto de algún *ConcreteProduct*

Factory method
Consecuencias

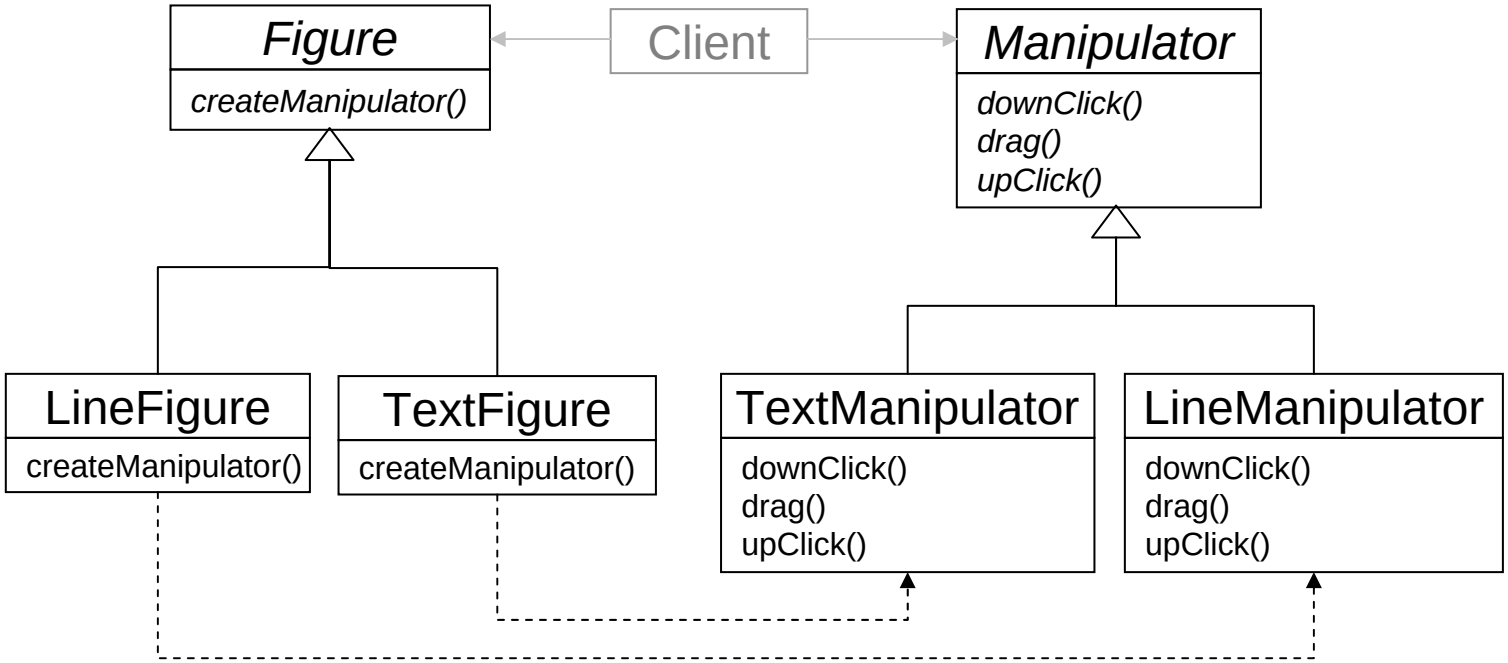
- Elimina la necesidad de ligar clases específicas de una aplicación al código, que trata con la interfaz *Product* y funciona con cualquier clase *ConcreteProduct*

- El cliente debe crear clases hijas de *Creator* para cada tipo de producto concreto

- Proporciona enganches para las subclases
  - La creación de objetos con métodos factoría es más flexible
  - Las subclases pueden dar una versión extendida del código padre

Factory method
Consecuencias

- Conecta jerarquías de clases paralelas (delegación)



Factory method
Implementación

- Existen dos variantes principales:
  - *Creator* es una clase abstracta y no implementa el método factoría
  - *Creator* es concreta y proporciona una implementación por defecto

- Métodos factoría parametrizados: crean varios tipos de producto, identificados por un parámetro del método

```
public class Creator {
   public Product factoryMethod (ProductId id) {
     if (id==MINE)  return new ConcreteProductA();
     if (id==YOURS) return new ConcreteProductB();
     return null;
}}
public class MyCreator extends Creator {
   public Product factoryMethod (ProductId id) {
     if (id==MINE)   return new ConcreteProductB();
     if (id==YOURS)  return new ConcreteProductA();
     if (id==THEIRS) return new ConcreteProductC();
     return super.factoryMethod(id);
}}
```

Factory method

Código de ejemplo: laberinto


```
public class MazeGame {
   // factory methods
   Maze makeMaze () { return new Maze(); }
   Wall makeWall () { return new Wall(); }
   Room makeRoom (int n) { return new Room(n); }
   Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }

   // create maze
   Maze createMaze () {
      Maze aMaze = makeMaze();
      Room r1 = makeRoom(1), r2 = makeRoom(2);
      Door aDoor = makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(Direction.NORTH, makeWall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, makeWall());
      r1.setSide(Direction.WEST,  makeWall());
      r2.setSide(Direction.NORTH, makeWall());
      r2.setSide(Direction.EAST,  makeWall());
      r2.setSide(Direction.SOUTH, makeWall());
      r2.setSide(Direction.WEST,  aDoor);
      return aMaze;
}}
```

Factory method

Código de ejemplo: laberinto


```
// Podemos crear nuevos tipos de laberinto


public class BombedMazeGame extends MazeGame {
   Wall makeWall ()      { return new BombedWall(); }
   Room makeRoom (int n) { return new RoomWithABomb(n); }
}


public class EnchantedMazeGame extends MazeGame {
   Room makeRoom (int n) { return new EnchantedRoom(n, castSpell()); }
   Door makeDoor (Room r1, Room r2) { return new DoorNeedingSpell(r1, r2); }
   protected Spell castSpell() { ... }
}
```

Abstract factory

Propósito

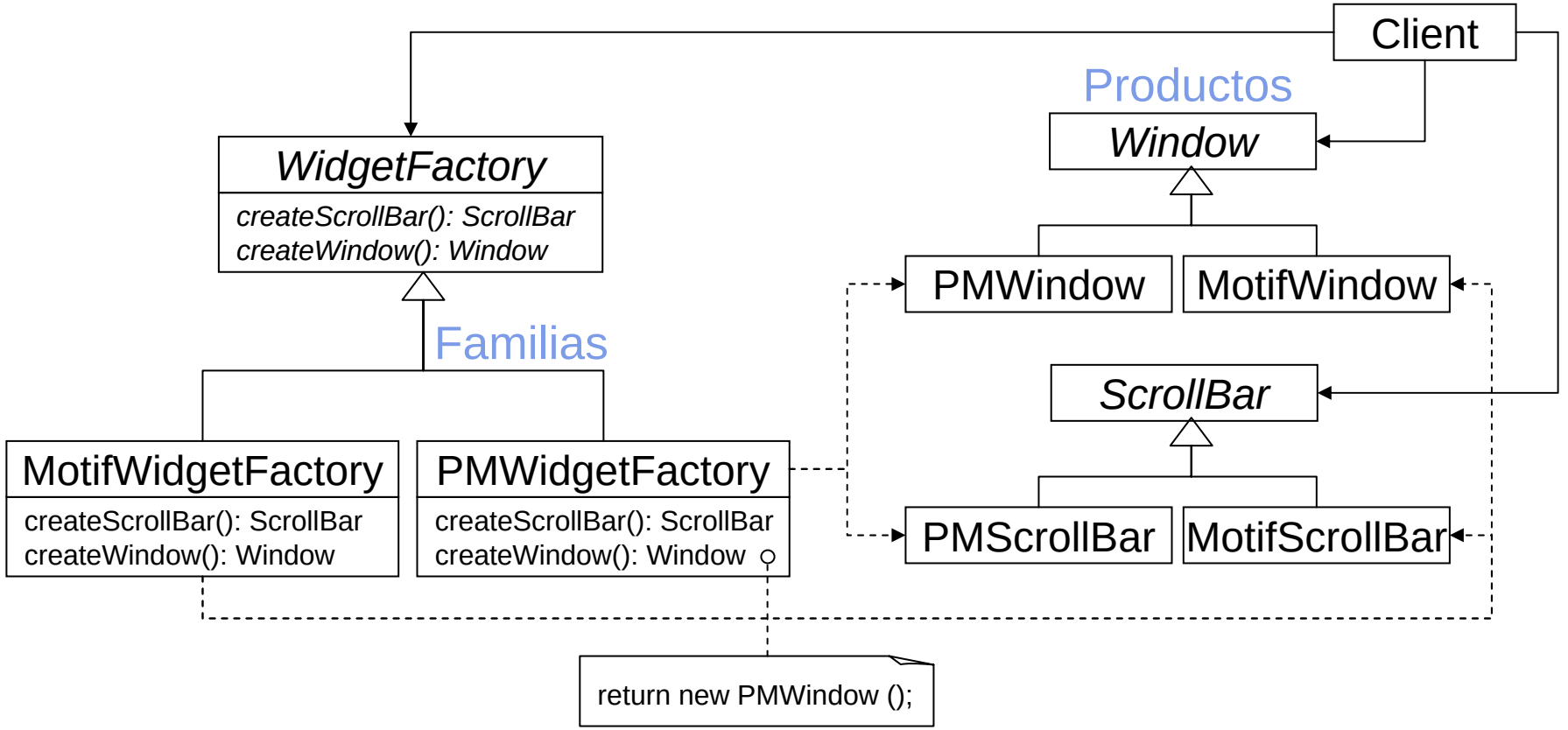
- Define una interfaz para crear familias de objetos relacionados o dependientes, sin especificar sus clases concretas

- También conocido como *kit*

Abstract factory
Motivación

- Ej: un framework para la construcción de interfaces de usuario que permita varios *look & feel* (ej. *Presentation Manager* y *Motif*)
- El cliente no debe cambiar porque cambie la interfaz de usuario
- Solución:
  - El cliente trabaja con las clases abstractas, independientemente del *look & feel* concreto
  - Una clase abstracta *WidgetFactory* con la interfaz para crear cada tipo de *widget*. Subclases concretas para cada *look & feel*
  - Una clase abstracta para cada tipo de *widget*. Subclases para cada *look & feel*

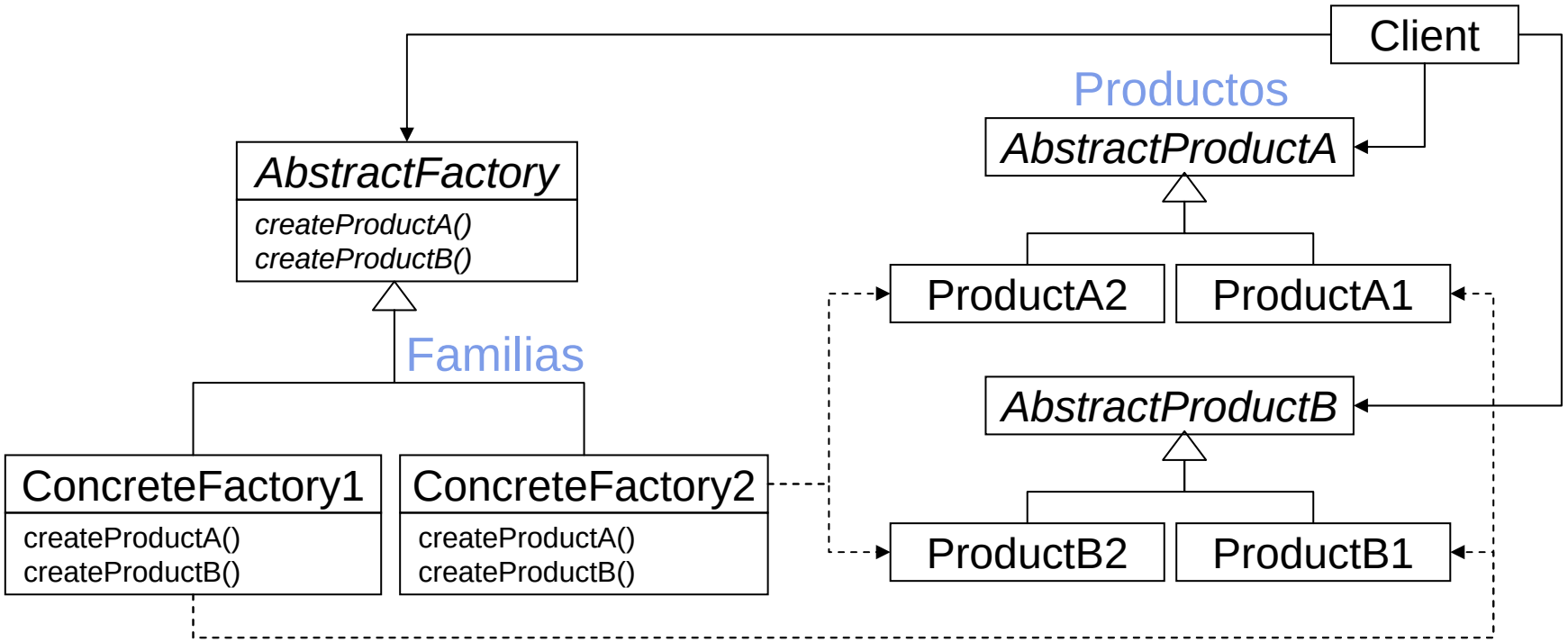
Abstract factory
Motivación



Abstract factory
Aplicabilidad

- Usa el patrón *Abstract factory* cuando:
  - Un sistema debe ser independiente de cómo se crean, componen y representan sus productos
  - Un sistema debe configurarse con una de entre varias familias de productos
  - Una familia de productos relacionados están hechos para usarse juntos, y se necesita cumplir esa restricción
  - Se desea ofrecer una biblioteca de clases-producto, revelando sus interfaces pero no sus implementaciones

Abstract factory
Estructura



Abstract factory
Participantes

- **AbstractFactory** (*WidgetFactory*): define la interfaz para crear objetos producto abstractos
- **ConcreteFactory** (*MotifWidgetFactory*, *PMWidgetFactory*): implementa las operaciones para crear objetos producto concretos
- **AbstractProduct** (*Window*, *ScrollBar*): define la interfaz de un tipo de objeto producto
- **ConcreteProduct** (*MotifWindow*, *MotifScrollBar*)*:*
  - define un objeto producto a crear con la factoría concreta correspondiente
  - implementa la interfaz *AbstractProduct*
- **Client***:* sólo usa las interfaces de *AbstractFactory* y *AbstractProduct*

Abstract factory
Consecuencias

- Aísla al cliente de las clases concretas (implementación)
  - Ayuda a controlar la clase de objetos que crea una aplicación
- Permite cambiar fácilmente de familia de productos
- Promueve la consistencia entre productos (esto es, que una aplicación utilice objetos de una sola familia a la vez)
- La inclusión de nuevos tipos de producto es difícil

Abstract factory
Implementación

- Factorías como *Singleton*
  - Asegura una sola instancia de factoría concreta por familia

- ¿Cómo crear los productos?
  - Utilizando un *factory method* por producto

Abstract factory

Código de ejemplo: laberinto


```
// factoría abstracta (proporciona una implementación por defecto)
public class MazeFactory {
   Maze makeMaze () { return new Maze(); }
   Wall makeWall () { return new Wall(); }
   Room makeRoom (int n) { return new Room(n); }
   Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }
}
public class MazeGame {
   // @param MazeFactory factory: factoría a usar para la creación de componentes
   Maze createMaze (MazeFactory factory) {
      Maze aMaze = factory.makeMaze();
      Room r1 = factory.makeRoom(1), r2 = factory.makeRoom(2);
      Door aDoor = factory.makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(Direction.NORTH, factory.makeWall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, factory.makeWall());
      r1.setSide(Direction.WEST,  factory.makeWall());
      r2.setSide(Direction.NORTH, factory.makeWall());
      r2.setSide(Direction.EAST,  factory.makeWall());
      r2.setSide(Direction.SOUTH, factory.makeWall());
      r2.setSide(Direction.WEST,  aDoor);
      return aMaze;
 }}
```

Abstract factory

Código de ejemplo: laberinto


```
// factorías concretas (sobrescriben métodos de la factoría abstracta)

public class BombedMazeFactory extends MazeFactory {
   Wall makeWall ()      { return new BombedWall(); }
   Room makeRoom (int n) { return new RoomWithABomb(n); }
}

public class EnchantedMazeFactory extends MazeFactory {
   Room makeRoom (int n) { return new EnchantedRoom(n, castSpell()); }
   Door makeDoor (Room r1, Room r2) {return new DoorNeedingSpell(r1, r2); }
   protected Spell castSpell() { }
}

// cliente

public class MazeTest {
   public static void main (String args[]) {
      MazeGame game = new MazeGame();
      MazeFactory factory = new BombedMazeFactory();
      game.createMaze(factory);
   }
}
```

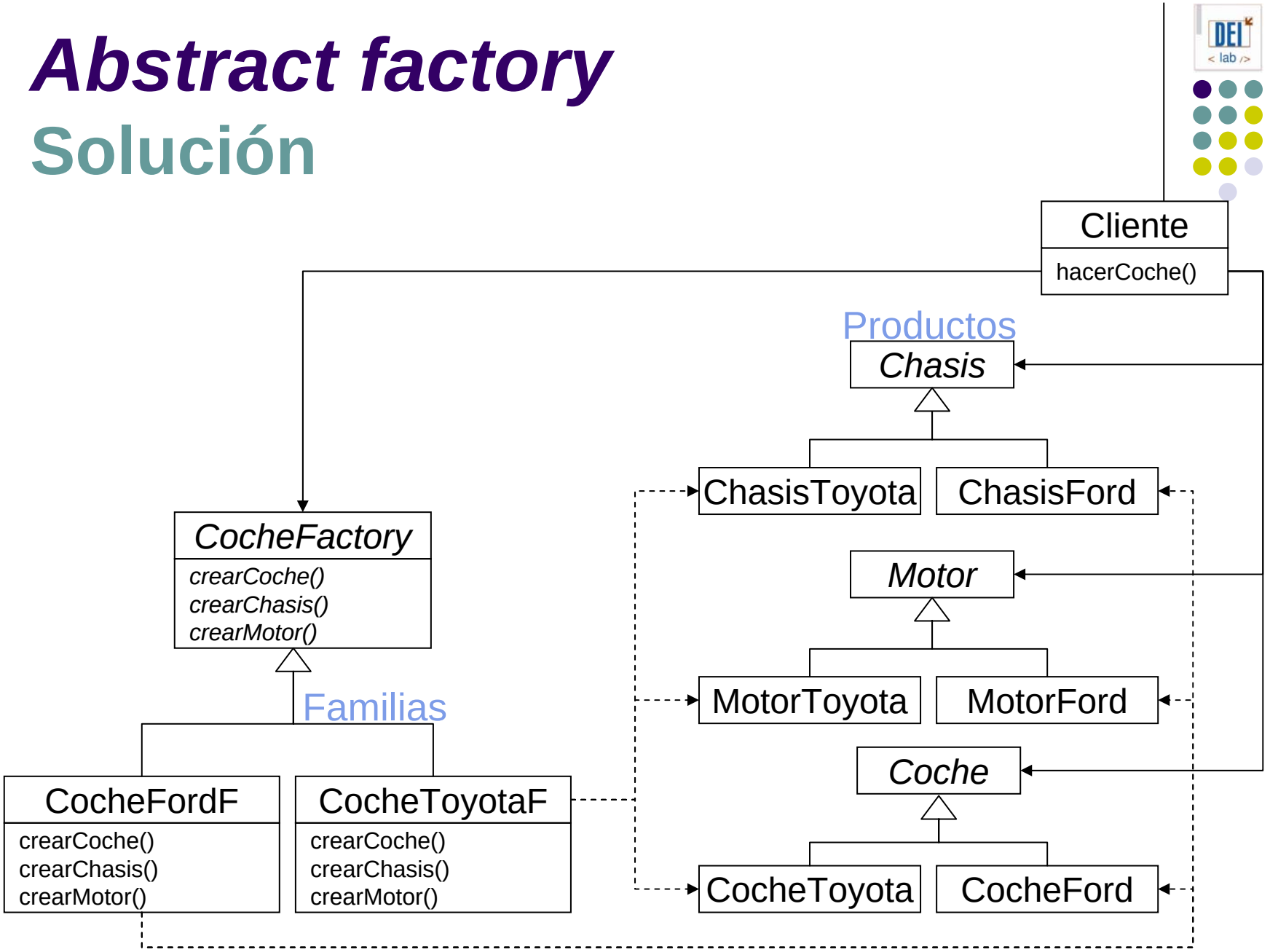
Abstract factory
Ejercicio

- Construir una aplicación para construir coches mediante el ensamblado de sus partes (motor, chasis, etc.)
  - Los componentes de un coche deben tener la misma marca
  - Hay múltiples marcas (Ford, Toyota, Opel...)
  - Es responsabilidad del cliente ensamblar las piezas

- Definir el diseño de la aplicación
- Definir el código necesario para construir un coche

Abstract factory

Solución



Abstract factory
Solución


```
public abstract class CocheFactory {
   abstract Coche  crearCoche();
   abstract Motor  crearMotor();
   abstract Chasis crearChasis();
}
public class CocheFordF {
   Coche  crearCoche()  { return new CocheFord();  }
   Motor  crearMotor()  { return new MotorFord();  }
   Chasis crearChasis() { return new ChasisFord(); }
}
public class Cliente {
   public static void main (string args[]) {
      Coche coche;
      String tipo;
      tipo = leerTipoDesdeTeclado();
      if (tipo.equals("toyota")) coche = hacerCoche(new CocheToyotaF());
      if (tipo.equals("ford"))   coche = hacerCoche(new CocheFordF  ());
   }
   Coche hacerCoche(CocheFactory factoria) {
      Coche c = factoria.crearCoche();
      c.addMotor (factoria.crearMotor());
      c.addChasis(factoria.crearChasis());
      return c;
   }
}
```

Singleton

Propósito

- Asegurar que una clase tiene una única instancia y proporciona un punto de acceso global a la misma

Motivación

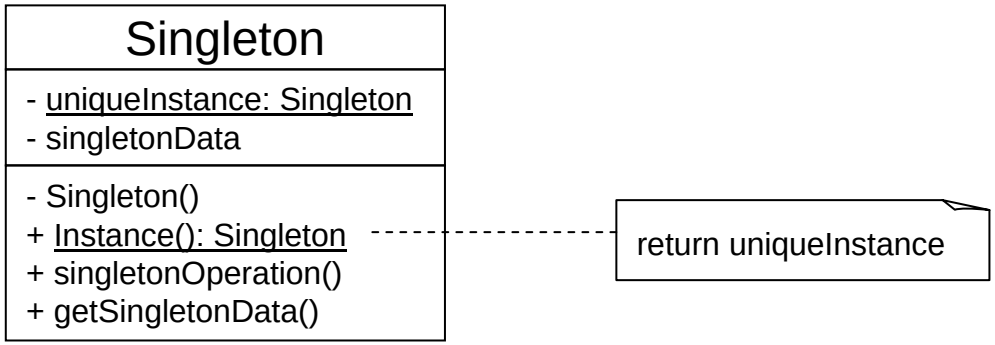
- A veces es importante asegurar que una clase sólo tiene una instancia (por ejemplo una sola cola de impresión, un gestor de ventanas, un sistema de ficheros...)

- Solución:
  - Una variable global: no, ya que no impide crear múltiples objetos
  - Responsabilidad del constructor de la clase

Singleton
Aplicabilidad

- Usa el patrón *Singleton* cuando:
  - Debe haber exactamente una instancia de una clase, que debe ser accesible a los clientes a través de un punto de acceso conocido
  - La instancia única pueda tener subclases, y los clientes deban ser capaces de usar las subclases sin modificar su propio código

Singleton
Estructura



Participantes

- **Singleton***:*
  - Define una operación *Instance* estática que permite a los clientes acceder a su instancia única
  - Puede ser responsable de crear su única instancia

Singleton
Consecuencias

- Acceso controlado a la instancia única
- Espacio de nombres reducido (mejora sobre el uso de variables globales)
- Permite refinamiento de operaciones (mediante subclases)
- Permite un número variable de instancias (cambiando la operación de acceso a la instancia *singleton*)
- Es más flexible que los métodos (estáticos) de clase
  - ¿cómo permitir más de una instancia de la clase?

Singleton
Implementación


```
public class Singleton {
   private static Singleton instance = null;
   protected Singleton() {
      // este método se define para evitar
      // la creación de la clase con new
   }
   public static Singleton Instance () {
      // lazy instantiation
      if (instance==null)
         instance = new Singleton();
      return instance;
   }
}

// código cliente

Singleton miSingleton = Singleton.Instance();
Singleton tuSingleton = Singleton.Instance();
```

Singleton
Implementación

- **Subclasificación***:*
  - Especificar en *Instance* qué subclase concreta instanciar

```
public class Singleton {
   private static Singleton instance;
   protected Singleton () { }
   public static Singleton Instance (String tipo) {
      if (instance==null) {
         if (tipo.equals("UNO")) instance = new SingletonUno();
         else if ...
      }
      return instance;
   }
   ...
}
```

Constructor protegido

Constructor de las subclases público

Singleton
Implementación

- **Subclasificación***:*
  - Mover la implementación de *Instance* a la subclase

```
public class Singleton {
   protected static Singleton instance;
   protected Singleton () { }
   public static Singleton Instance () {
      if (instance==null) instance = new Singleton();
      return instance;
   }
}

public class SingletonUno extends Singleton {
   public static Singleton Instance () {
      if (instance==null) instance = new SingletonUno();
      return instance;
   }
}
```

Atributo protegido

En java no se pueden sobrescribir los métodos estáticos !!!

Singleton
Implementación

- **Subclasificación:**
  - Registrar los objetos de las subclases en *Singleton*

```
public class Singleton {
   private static Hashtable<Singleton> h = new Hashtable<Singleton>();
   private static Singleton instance;
   protected Singleton () { }
   public static void register (String tipo, Singleton s) { h.put(tipo, s); }
   public static Singleton Instance (String tipo) {
      if (instance==null) instance = h.get(tipo);
      return instance;
   }
}
public class SingletonUno extends Singleton {
   private static SingletonUno su;
   protected SingletonUno () { }
   public static void register () {
      if (su==null) {
         su = new SingletonUno();
         register("SingletonUno", su);
      }
   }
}
// cliente
SingletonUno.register();
Singleton s = Singleton.Instance("SingletonUno");
```

Un único objeto. Para un objeto de cada clase: `return h.get(tipo);`

Las subclases se registran en la clase padre

Singleton
Código de ejemplo: laberinto


```
public class MazeFactory {
   private static MazeFactory _instance = null;
   protected MazeFactory () {
   }
   static MazeFactory Instance () {
      if (_instance==null)
         _instance = new MazeFactory();
      return _instance;
   }
   static MazeFactory Instance (String style) {
      if (_instance==null) {
         if (style.equals(“bombed”)
            _instance = new BombedMazeFactory();
         else if (style.equals(“enchanted”))
            _instance = new EnchantedMazeFactory();
         else
            _instance = new MazeFactory();
      }
      return _instance;
   }
   // métodos make*
   // ...
}
```