Tema 4
Patrones de Diseño

Análisis y Diseño de Software

2º Ingeniería Informática

Universidad Autónoma de Madrid

Indice

- **Introducción.**
  - **Patrones.**
  - **Descripción de los Patrones.**
  - **Ejemplo: Patrones en el MVC de Smalltalk.**
  - **El catálogo de patrones.**
- Patrones de Creación.
- Patrones Estructurales.
- Patrones de Comportamiento.
- Conclusiones.
- Bibliografía.

Introduccción

- Diseño específico para el problema, pero general como para poder adecuarse a futuros requisitos y problemas.
- Evitar el rediseño en la medida de lo posible.
- Evitar resolver cada problema partiendo de cero.
- Reutilizar soluciones que han sido útiles en el pasado.
- Patrones recurrentes de clases y comunicación entre objetos en muchas soluciones de diseño.

Patrón

- Un esquema que se usa para solucionar un problema.
- El esquema ha sido probado extensivamente, y ha funcionado. Se tiene experiencia sobre su uso.
- Existen en muchos dominios:
  - Novelas y el cine: “héroe trágico”, “comedia romántica”, etc.
  - Arte.
  - Ingenierías.
  - Arquitectura.
    - Christopher Alexander. “*A Pattern Language: Towns, Buildings, Construction”. 1977.*
    - http://www.patternlanguage.com

Patrones de Diseño

- Reutilizar diseños abstractos que no incluyan detalles de la implementación.
- Un patrón es una descripción del problema y la esencia de su solución, que se puede reutilizar en casos distintos.
- Es una solución adecuada a un problema común.
- Asociado a orientación a objetos, pero el principio general es aplicable a todos los enfoques de diseño software.

Patrones de Diseño

- Documentar la experiencia en el diseño, en forma de un catálogo de patrones.

- Categorías de patrones:

  - **De creación**: implica el proceso de instanciar objetos.

  - **Estructurales**: composición de objetos.

  - **De comportamiento**: cómo se comunican los objetos, cooperan y distribuyen las responsabilidades para lograr sus objetivos.

Patrones de Diseño

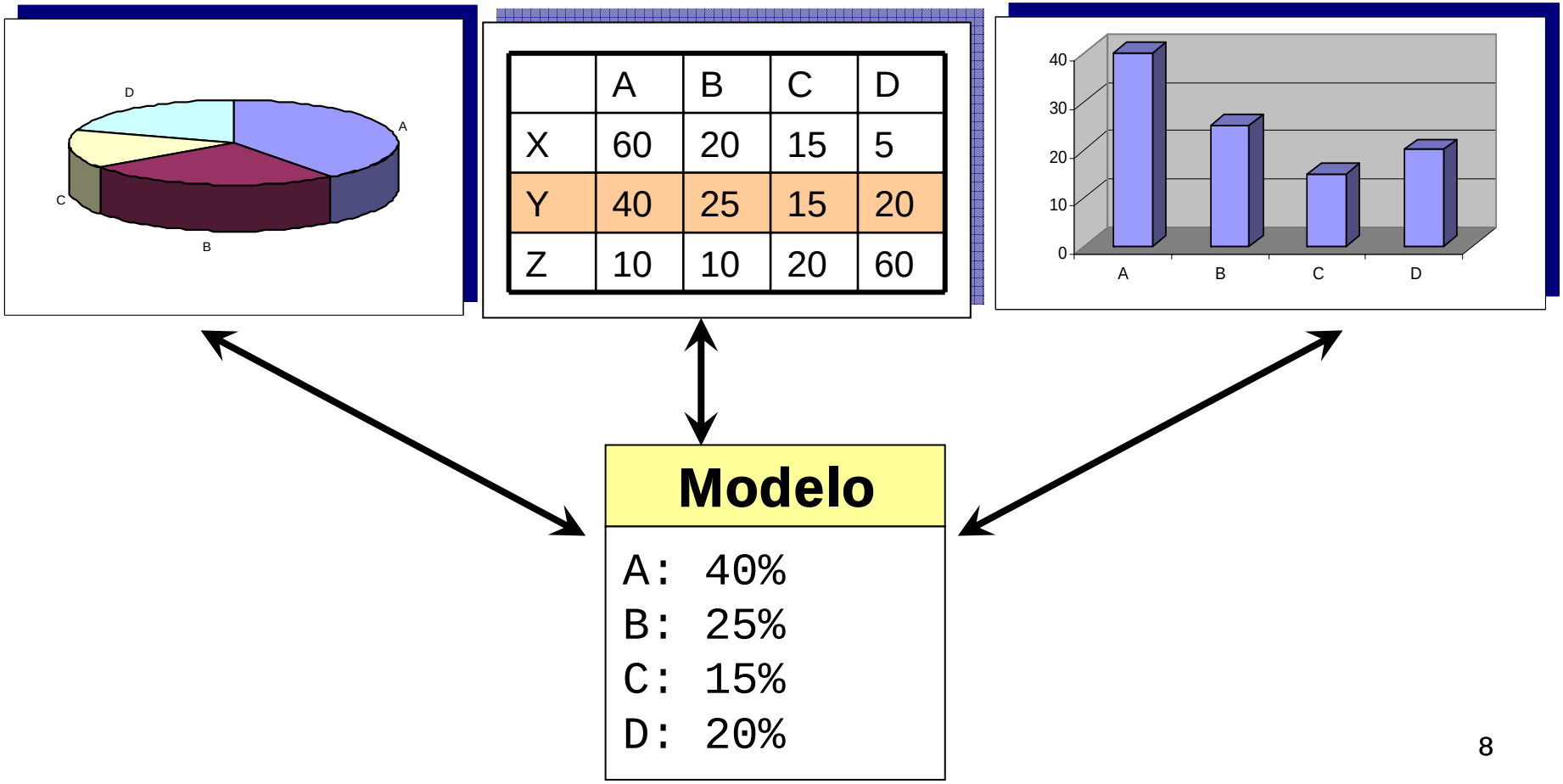
Estructura de un patrón

- ***Nombre del patrón.***
  - Describe el problema de diseño, junto con sus soluciones y consecuencias.
  - Vocabulario de diseño.
- ***Problema.***
  - Describe cuándo aplicar el patrón.
  - Explica el problema y su contexto.
- ***Solución.***
  - Elementos que forman el diseño, relaciones, responsabilidades.
  - No un diseño concreto, sino una plantilla que puede aplicarse en muchas situaciones distintas.
- ***Consecuencias.***
  - Resultados, ventajas e inconvenientes de aplicar el patrón.
  - P.ej.: relación entre eficiencia en espacio y tiempo; cuestiones de implementación, etc.

Patrones de Diseño

Ejemplo: MVC de Smalltalk

- Modelo/Vista/Controlador



Patrones de Diseño

Ejemplo: MVC de Smalltalk

- Separar los objetos con los datos (modelo), sus visualizaciones (vistas) y el modo en que la interfaz reacciona ante la entrada al usuario (controlador).

- Separar estos componentes, para aumentar la flexibilidad y reutilización.

- Desacoplar vistas de modelos, mediante un protocolo de subscripción/notificación.

- Cada vez que cambian los datos del modelo, avisar a las vistas que dependen de él. Estas se actualizan.

Patrones de Diseño

Ejemplo: MVC de Smalltalk

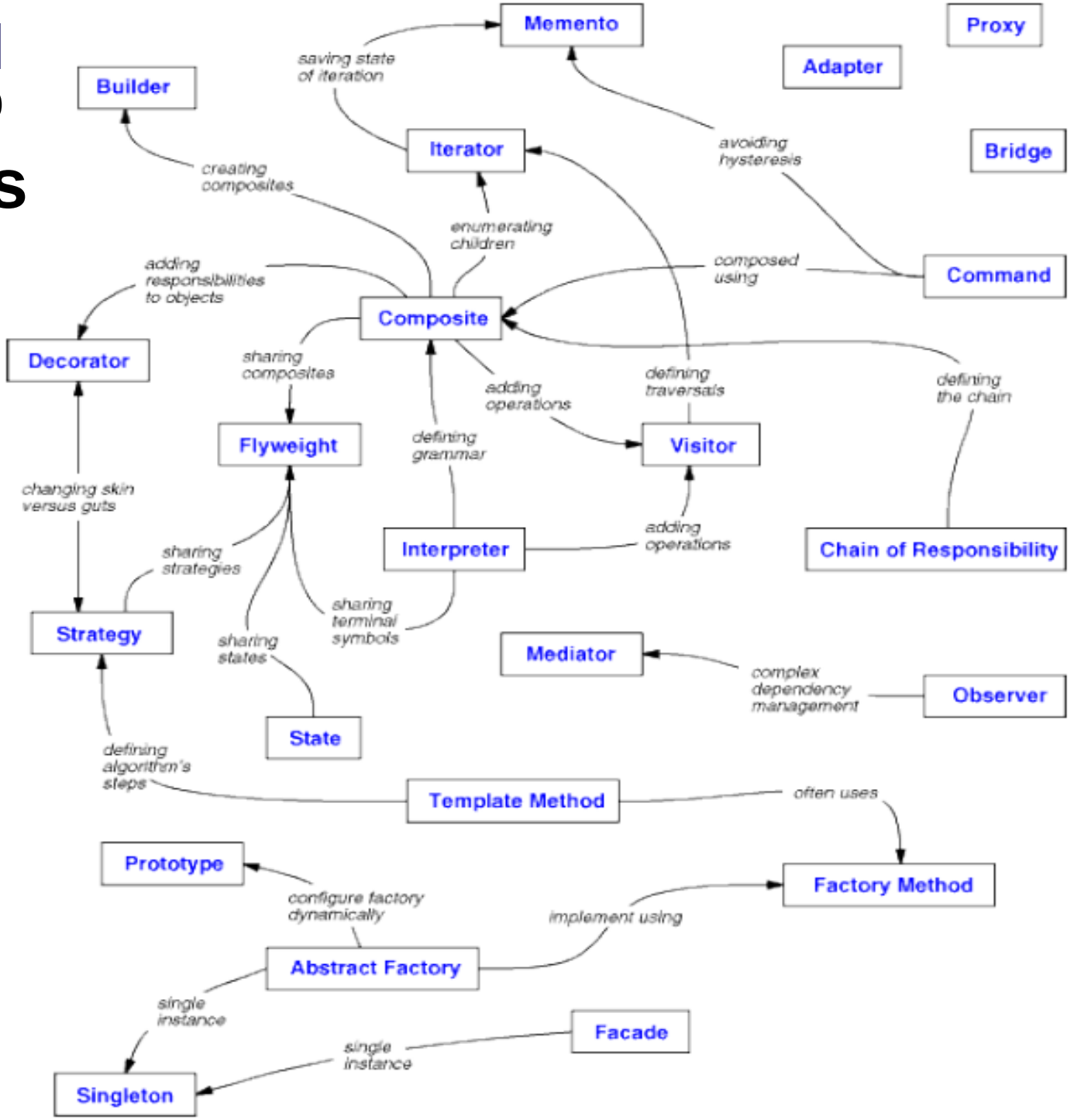
- Patrón *Observer*, más general (dependencias entre objetos).
- Las vistas se pueden anidar: Vistas compuestas y simples.
  - Generalización: Patrón *Composite*.
- La relación entre la vista y el controlador: patrón *Strategy* (un objeto que representa un algoritmo).
- Otros patrones:
  - *Factory Method*: especifica la clase controlador predeterminada para una vista.
  - *Decorator*: Añade capacidad de desplazamiento a una vista.

El catálogo de patrones

		Propósito		
		Creación	**Estructurales**	**De Comportamiento**
Ámbito	**Clase**	Factory Method	Adapter (de clases)	Interpreter. Template Method.
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (de objetos). Bridge. Composite. Decorator. Facade. Flyweight. Proxy.	Chain of Responsibility. Command. Iterator. Mediator. Memento. Observer. State. Strategy. Visitor.

El catálogo de patrones

Relaciones entre los patrones.



Indice

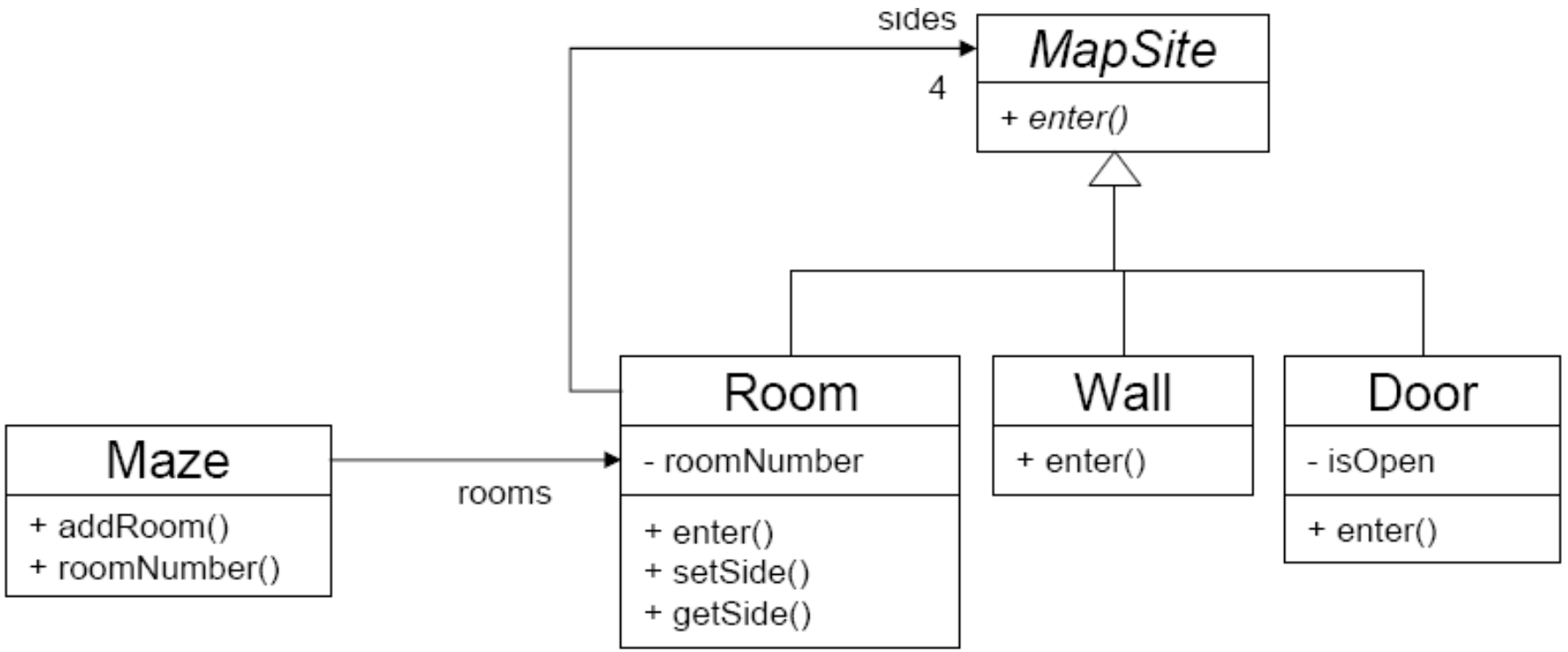
- Introducción.
- **Patrones de Creación.**
  - **Factory Method.**
  - **Abstract Factory.**
  - **Singleton.**
- Patrones Estructurales.
- Patrones de Comportamiento.
- Conclusiones.
- Bibliografía.

Patrones de Creación

- Abstraen el proceso de instanciación
- Ayudan a que el sistema sea independiente de cómo se crean, componen y representan los objetos.
- Flexibilizan:
  - qué se crea
  - quién lo crea
  - cómo se crea
  - cuándo se crea
- Permiten configurar un sistema:
  - estáticamente (compile-time)
  - dinámicamente (run-time)

Ejemplo

- Laberinto:



Ejemplo

```
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) { }
}
```

Ejemplo

```
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?

Ejemplo

```
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

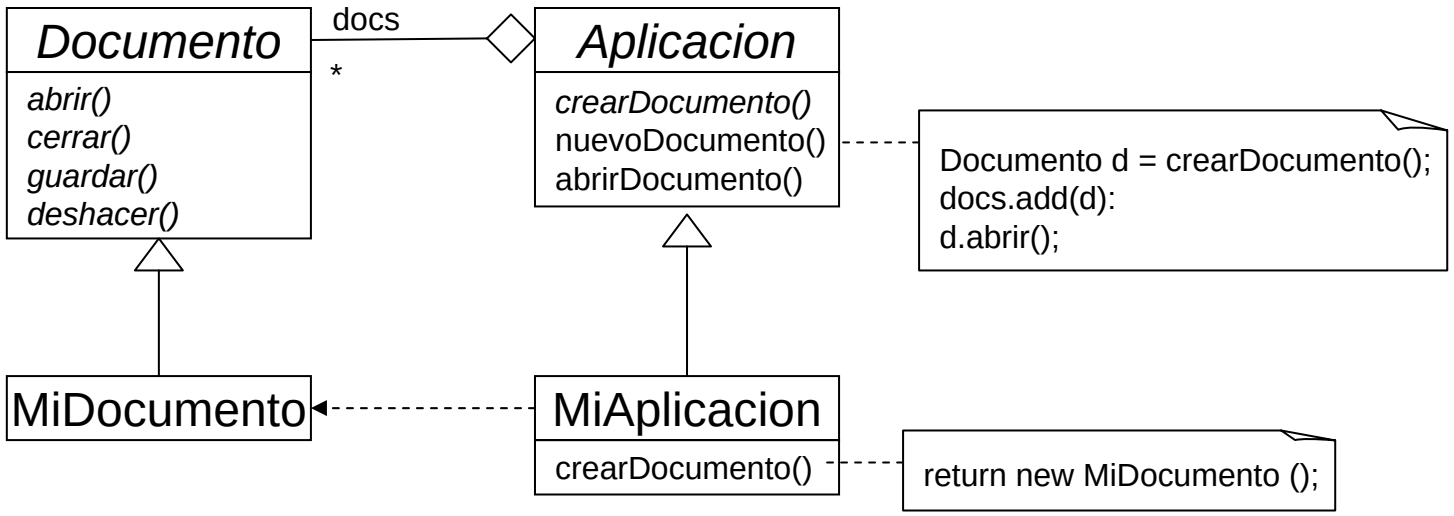
Factory Method

- **Propósito.**
  - Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
  - Permite que una clase delegue en sus subclases la creación de objetos.
- **También Conocido Como.**
  - Virtual Constructor.

Factory Method

- **Motivación.**
  - 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

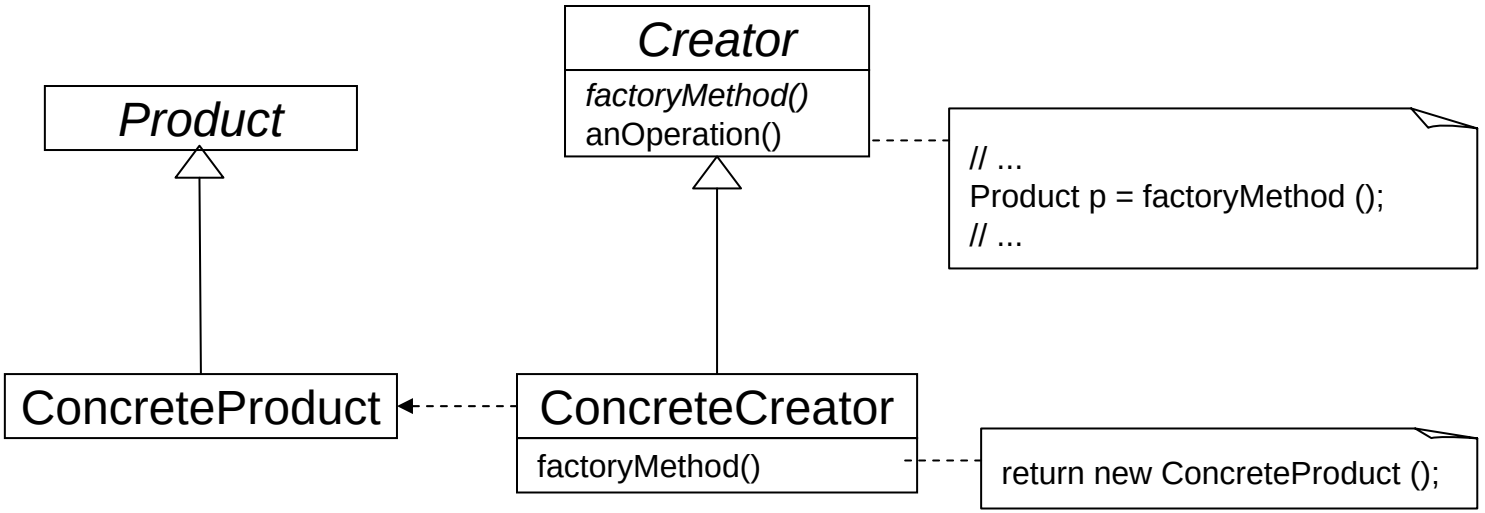
Factory Method



Factory Method

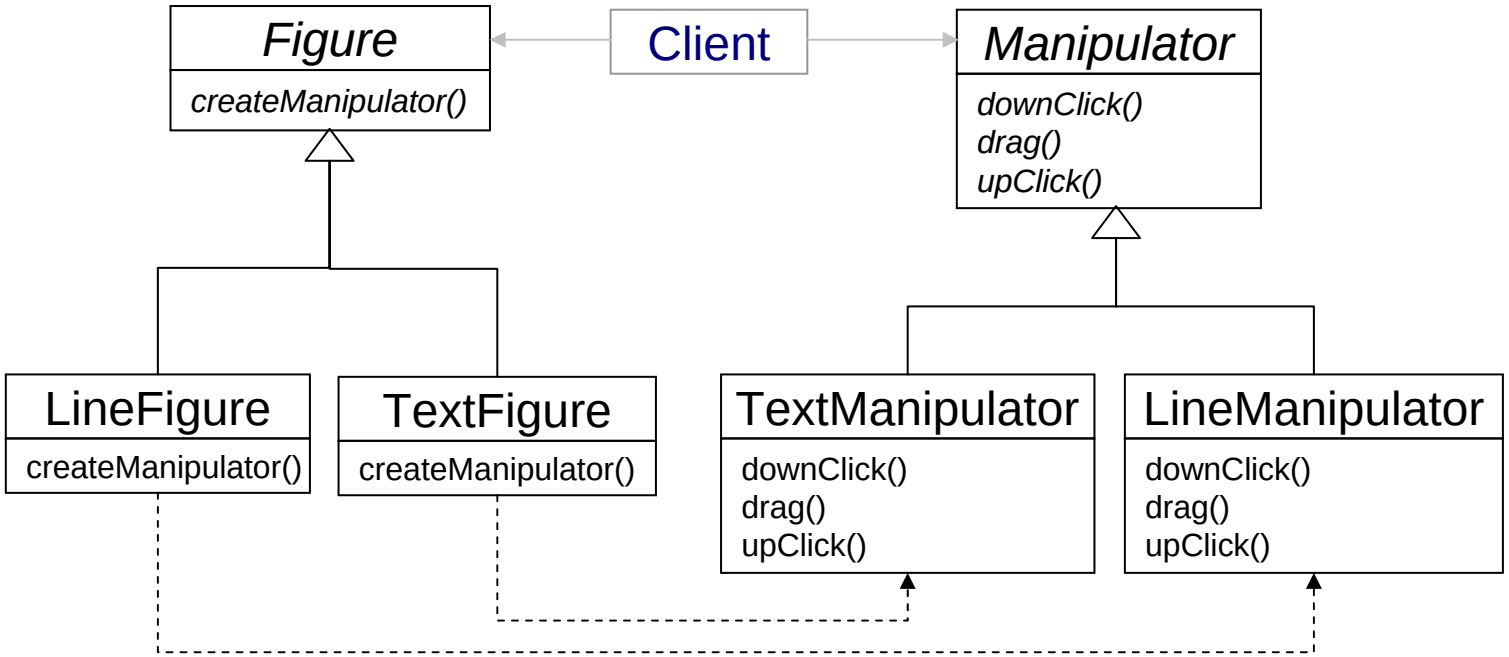
- Usa el patrón *Factory method* cuando:
  - Una clase no puede prever la clase de objetos que tiene que crear
  - Una clase quiere que sus subclases decidan qué objetos crean
  - Las clases delegan responsabilidades a una de entre varias subclases auxiliares, y queremos localizar en qué subclase concreta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

Estructura


Creator
factoryMethod()
anOperation()
// ...
Product p = factoryMethod ();
// ...
Product
ConcreteProduct
ConcreteCreator
factoryMethod()
return new ConcreteProduct ();

Consecuencias

- Conecta jerarquías de clases paralelas (delegación)



Ejemplo

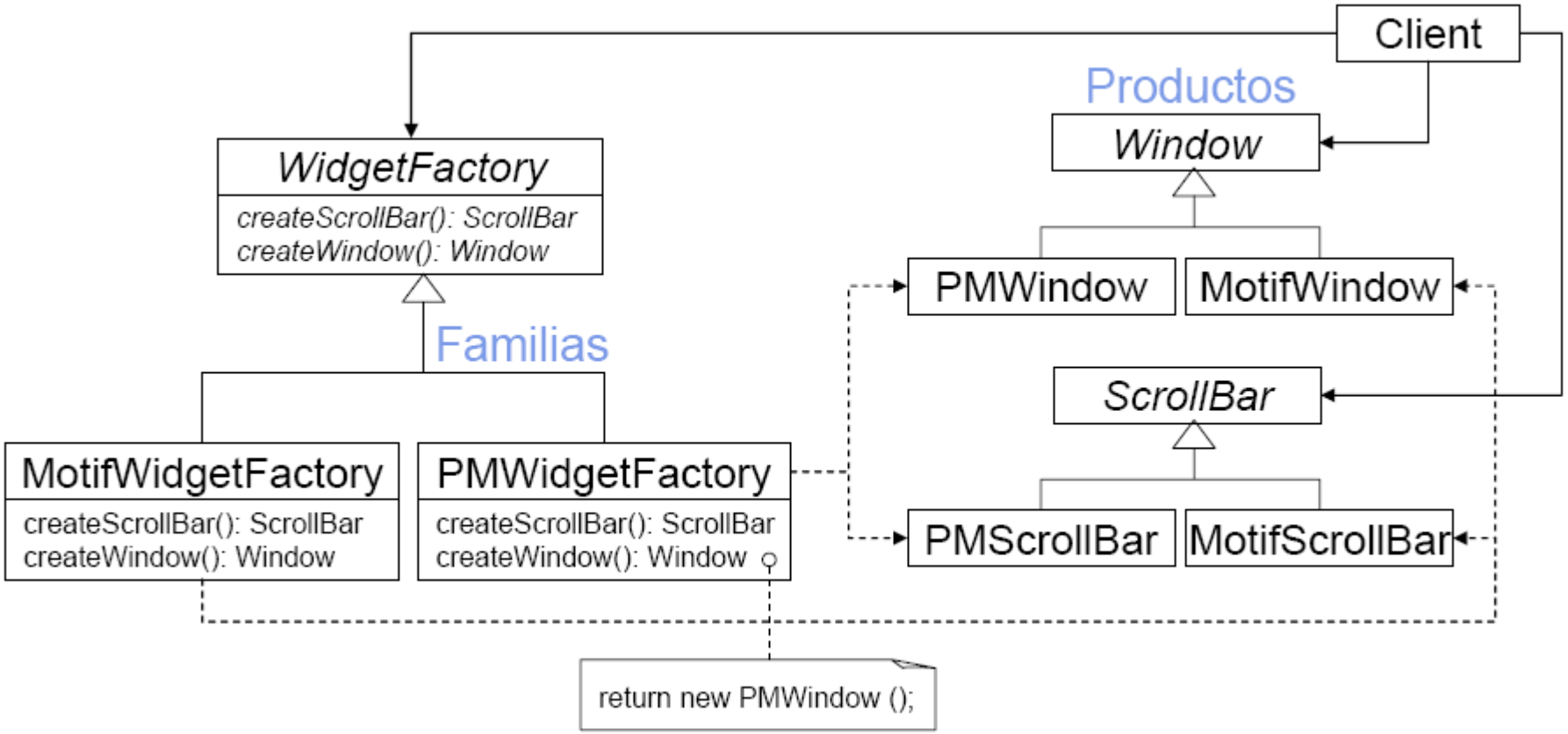
```
public class MazeGame {
   // factory methods
   Maze makeMaze () { return new Maze(); }
   Wall makeWall () { return new Wall(); }
   Room makeRoom (int n) { return new Room(n); }
   Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }

   // 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;
}}
```

Abstract Factory

- **Propósito.**
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
- **También Conocido Como.**
  - Kit.
- **Motivación.**
  - Ej.: un framework para la construcción de interfaces de usuario que permita varios "*look and feel*" (ej.: Motif y Presentation Manager).
    - Una clase abstracta *WidgetFactory* con la interfaz para crear cada tipo de widget.
    - Una clase abstracta para cada tipo de widget. Subclases concretas que implementan cada widget concreto.
    - De esta manera, los clientes son independientes del *look and feel* concreto.

Abstract Factory

- Motivación

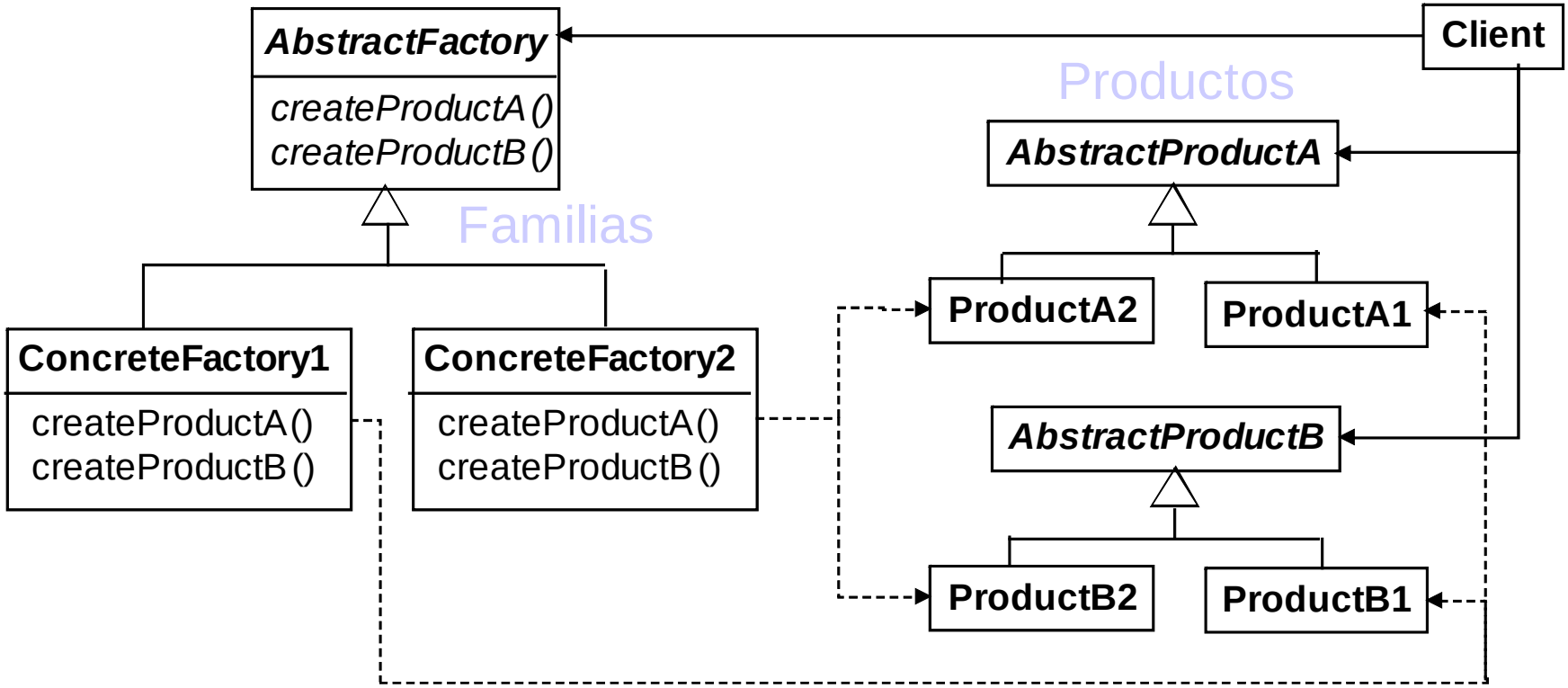


Abstract Factory

- **Aplicabilidad.** Usar este patrón cuando:
  - un sistema que deba ser independiente de cómo se crean, componen y representan sus productos.
  - un sistema que deba ser configurado con una familia de productos entre varias.
  - una familia de objetos producto relacionados que está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.
  - se quiere proporcionar una biblioteca de clases productos, y sólo se quiere revelar sus interfaces, no su implementación.

Abstract Factory

- Estructura:



Abstract Factory

- **Consecuencias.**
  - Aísla las clases concretas. Ayuda a controlar las clases de objetos que crea una aplicación. Aísla a los clientes de las clases de implementación.
  - Facilita el reemplazo de familias de productos.
  - Promueve la consistencia entre productos (que la aplicación use objetos de una sola familia a la vez).
  - Es difícil añadir un nuevo producto.

Ejemplo. *Laberinto.*

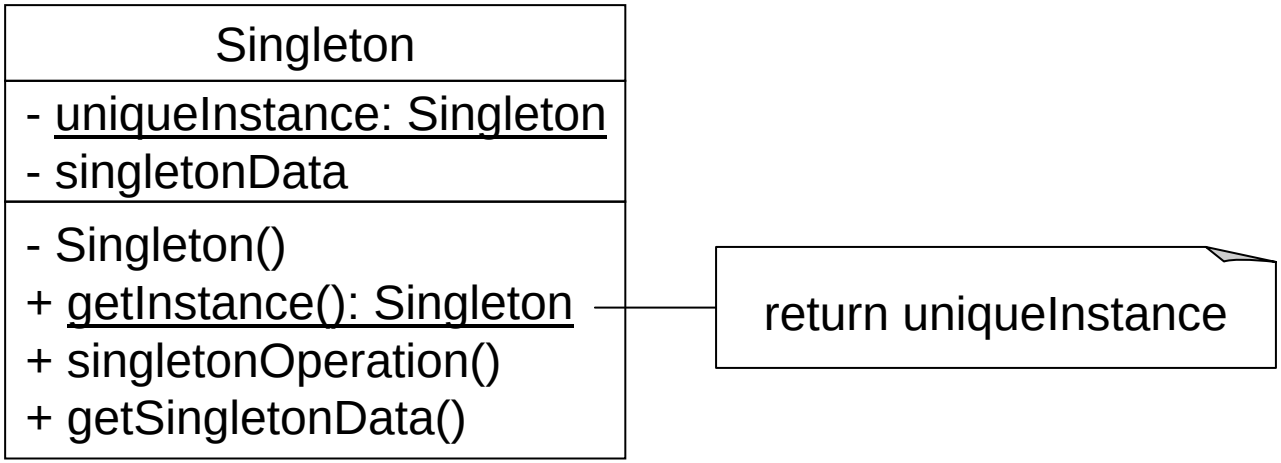
```
// factoría abstracta (proporciona una implementación por defecto)
public class MazeFactory {
  Maze makeMaze () { return new Maze(); }
  Wall makeWall () { return new Wall(); }
  Room makeRoom (int n) { return new Room(n); }
  Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }
}
public class MazeGame {
  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;
}}
```

Singleton

- **Intención.**
  - Asegurar que una clase tiene una única instancia y proporciona un punto de acceso global a dicha instancia.
- **Motivación.**
  - Hay veces que es importante asegurar que una clase tenga una sola instancia, por ejemplo:
    - Un gestor de ventanas
    - Una única cola de impresión
    - Un único sistema de ficheros
    - Un único fichero de log, o un único repositorio.
  - ¿Cómo asegurarlo? una variable global hace el objeto accesible, pero se puede instanciar varias veces.
  - Responsabilidad de la clase misma: actuar sobre el mensaje de creación de instancias

Singleton

- **Aplicabilidad.**
  - Cuando debe haber una sola instancia, y debe ser accesible a los clientes desde un punto de acceso conocido.
  - Cuando la única instancia debe ser extensible mediante subclasificación, y los clientes deben ser capaces de usar una instancia extendida sin modificar su código.
- **Estructura.**



Singleton

- **Participantes:**
  - Singleton:
    - Define una operación de clase, llamada “Instance” que deja a los clientes acceder a la única instancia.
    - Puede ser responsable de crear su única instancia.
- **Colaboraciones:**
  - Los clientes acceden a la instancia de un Singleton a través de la operación “getInstance()”.
- **Consecuencias:**
  - Acceso controlado a la única instancia.
  - Espacio de nombre reducido. Mejora sobre el uso de variables globales.
  - Permite el refinamiento de operaciones y la representación. Se puede subclasificar de la clase Singleton y configurar la aplicación con una instancia de esta clase.
  - Fácil modificación para permitir un número variable de instancias.
  - Más flexible que las operaciones de clase.

Singleton

- Esquema de implementación en Java

```
public class Singleton {
  private static final Singleton instance = new Singleton();
  // Constructor protegido para evitar instanciación desde fuera
  protected Singleton() {
     ...
  }
  public static Singleton getInstance() {
     return instance;
  }
}
```

Singleton

- Inicialización bajo demanda.

```
public class Singleton {
  private static Singleton instance = null;
  protected Singleton() {}

  public static Singleton getInstance () {
    // lazy instantiation
    if (instance==null) instance = new Singleton();
    return instance;
  }
}

// código cliente
Singleton miSingleton = Singleton.getInstance();
Singleton tuSingleton = Singleton.getInstance();
```

Singleton

Ejemplo. Laberinto.

```
public class MazeFactory {
  private static MazeFactory instance = null;
  protected MazeFactory () {}
  public static MazeFactory getInstance () {
    if (instance==null) instance = new MazeFactory();
    return instance;
  }
  public static MazeFactory getInstance (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*
  // ...
}
```

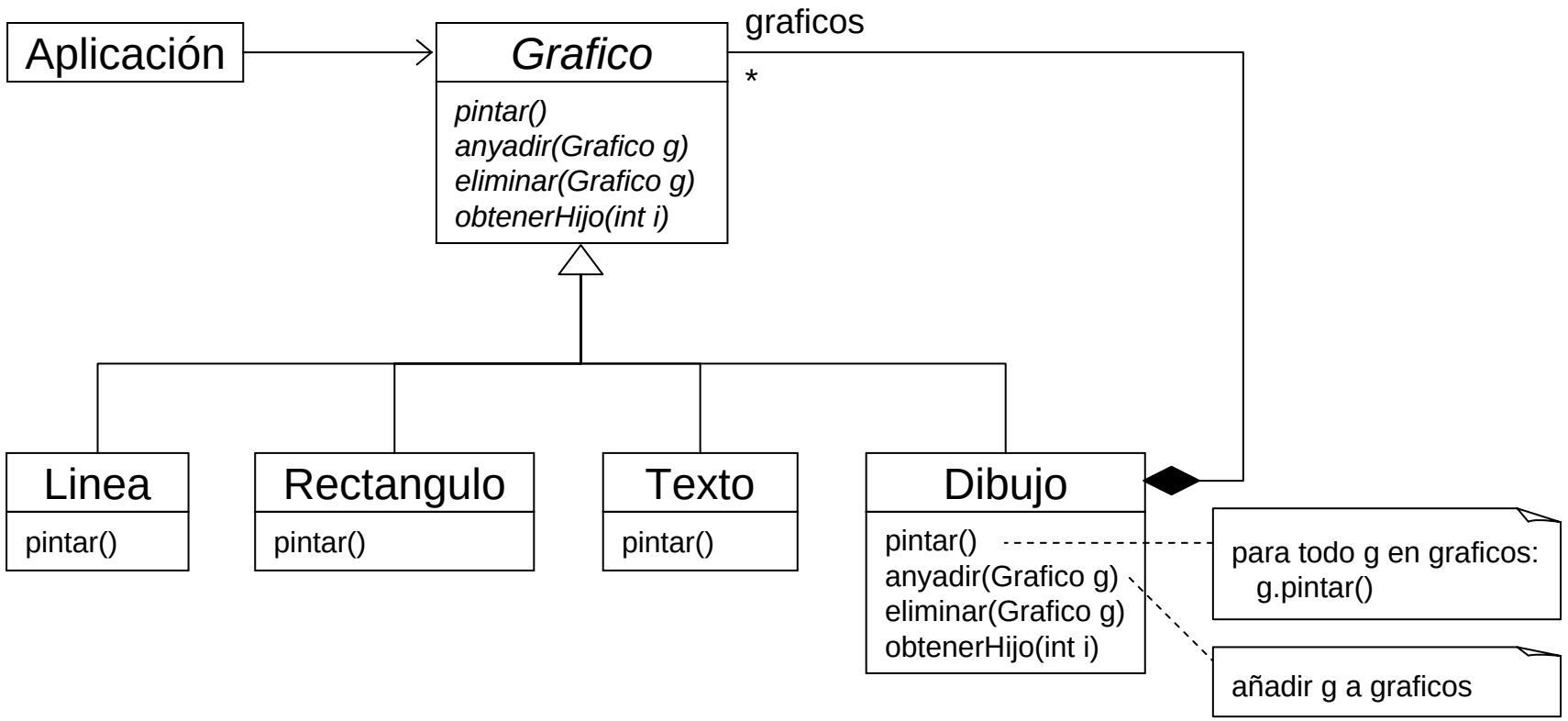
Indice

- Introducción.
- Patrones de Creación.
- **Patrones Estructurales.**
  - **Composite.**
  - **Proxy.**
- Patrones de Comportamiento.
- Conclusiones.
- Bibliografía.

Composite

- **Propósito:**
  - Componer objetos en estructuras arbóreas para representar jerarquías todo-parte. Manipular todos los objetos del árbol de manera uniforme
- **Motivación**
  - Ej.: las aplicaciones gráficas manejan grupos de figuras hechas de componentes sencillos (líneas, texto...)
- **Solución:**
  - Primitivas para los componentes sencillos, y otras para los contenedores? No porque no se tratan de manera uniforme
  - Definir una clase abstracta que represente componentes y contenedores, de la cual todas heredan, y que define sus operaciones

Composite

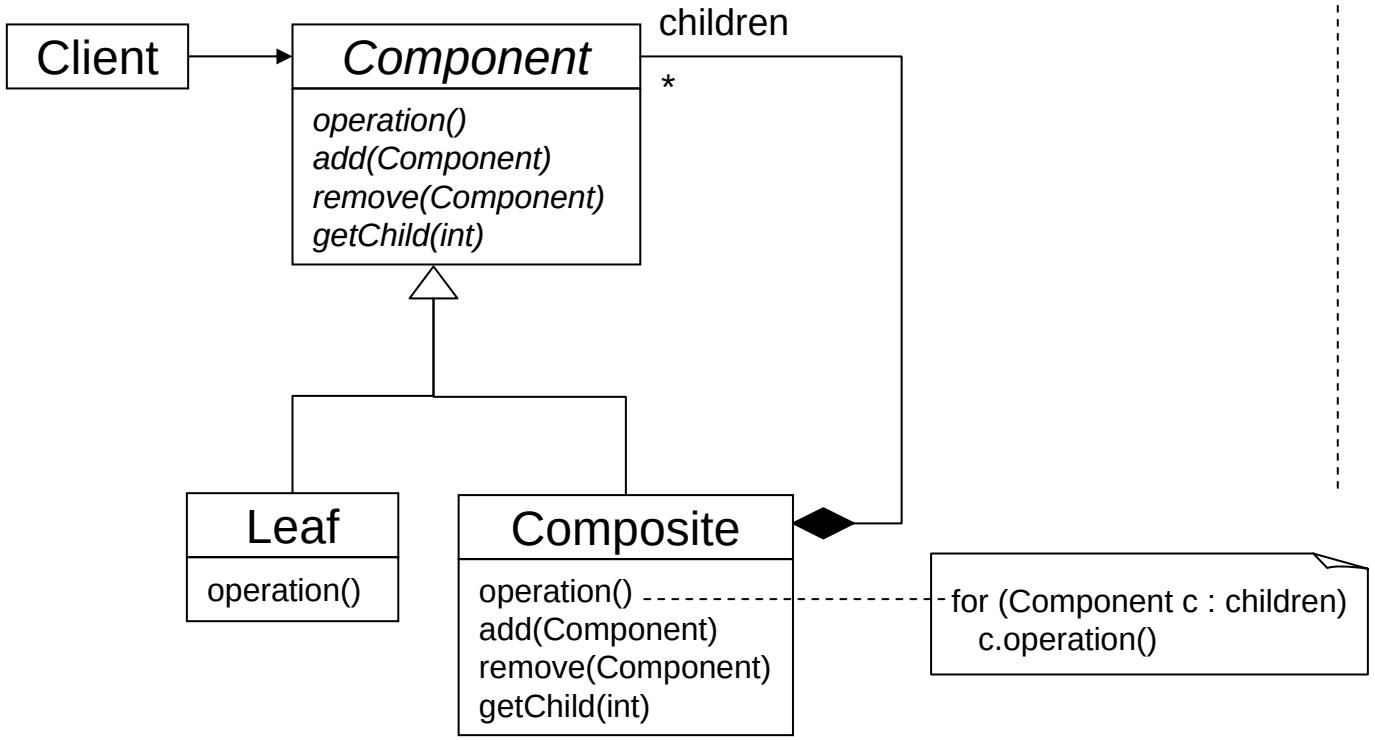

Aplicación
Grafico
pintar()
anyadir(Grafico g)
eliminar(Grafico g)
obtenerHijo(int i)
graficos
*
Linea
pintar()
Rectangulo
pintar()
Texto
pintar()
Dibujo
pintar()
anyadir(Grafico g)
eliminar(Grafico g)
obtenerHijo(int i)
para todo g en graficos:
g.pintar()
añadir g a graficos

Composite

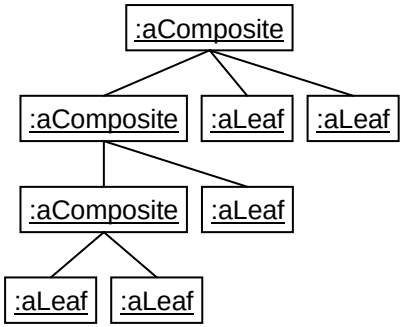
- Usa el patrón *Composite* cuando:
  - Quieres representar jerarquías de objetos todo-parte
  - Quieres ser capaz de ignorar la diferencia entre objetos individuales y composiciones de objetos. Los clientes tratarán a todos los objetos de la estructura compuesta uniformemente.

Composite

Estructura

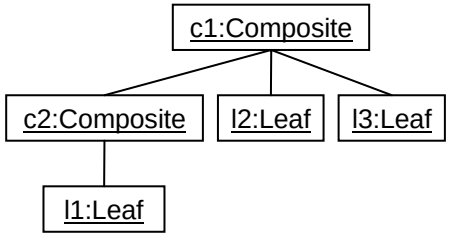


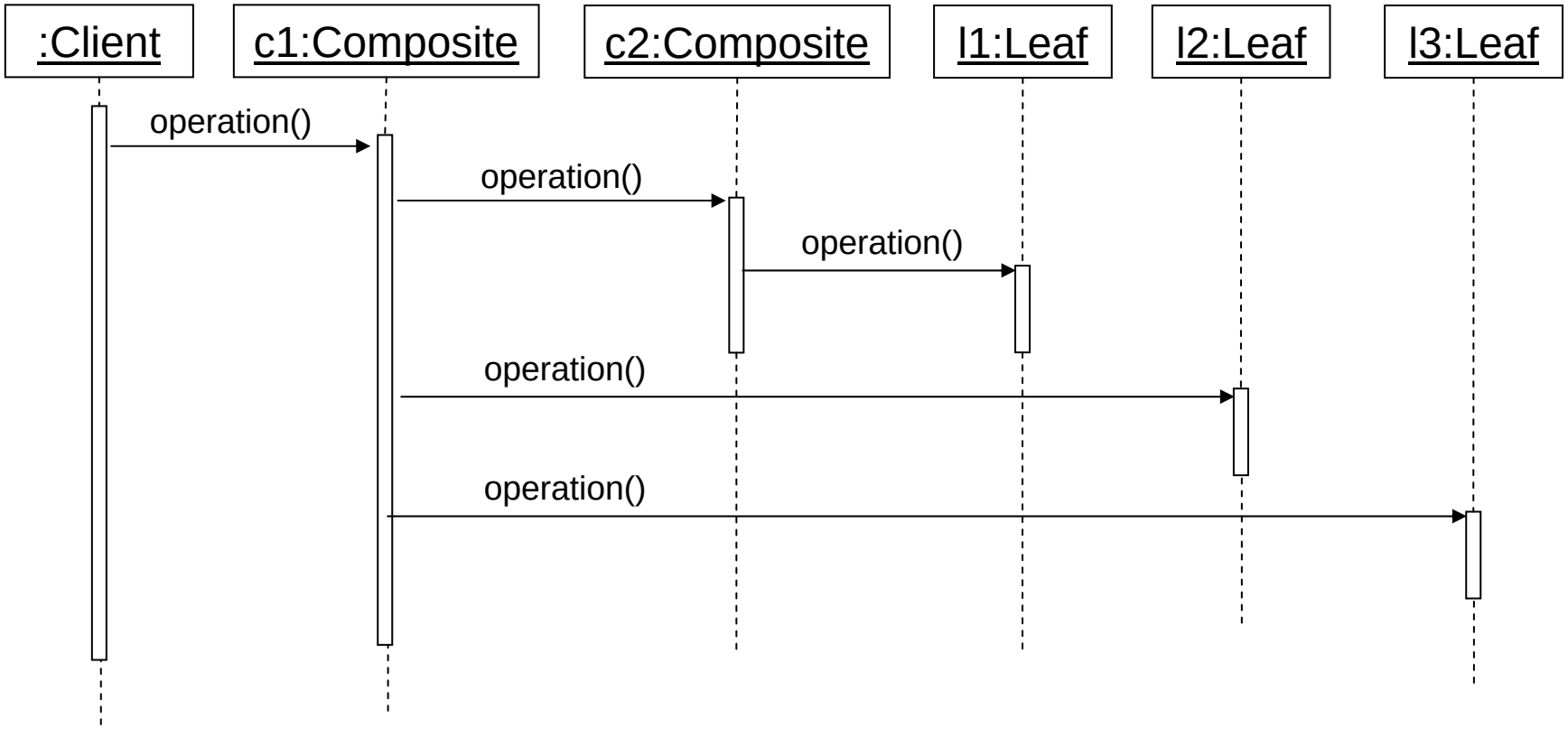
Ejemplo de estructura-objeto:



Composite

Colaboraciones. Ejemplo.

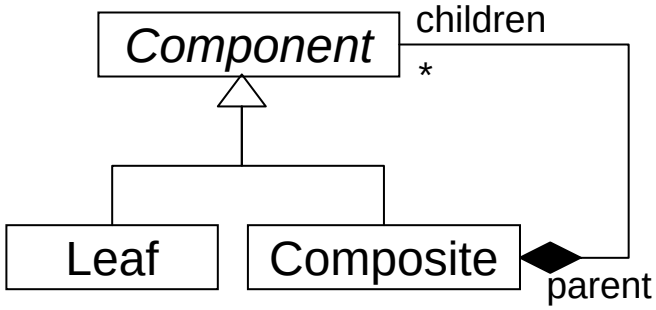




Composite

Implementación.

- Referencias explícitas a los padres
  - Simplifica algunas operaciones de la estructura compuesta
  - Definirlas en la clase *Component*
  - Gestionarlas al añadir/eliminar elementos de un *Composite*




```
public abstract class Component {
   protected Composite parent;
   public void setParent (Composite parent){
      this.parent = parent;
   }
   ...
}
public class Composite {
   protected List<Component> children;
   public void add(Component c) {
      children.add(c);
      c.setParent(this);
   }
...
}
```

- Compartir componentes
  - Útil para ahorrar memoria
  - La gestión de un componente con varios padres se complica

Composite

Código Ejemplo.

```
// Implementación con interfaces

public interface Component {
   public void add (Component c);
   public void remove (Component c);
   public Component getChild (int i);
}

public class Leaf implements Component {
   public void add (Component c)     {}  //también puede lanzar una excepción
   public void remove (Component c)  {}  //también puede lanzar una excepción
   public Component getChild (int i) { return null; }
}

public class Composite implements Component {
   private List<Component> children = new ArrayList<Component>();
   public void add (Component c)     { children.add (c); }
   public void remove (Component c)  { children.remove (c); }
   public Component getChild (int i) { return children.get(i); }
}
```

Composite

Código Ejemplo.

```
// Implementación con clase abstracta

public abstract class Component {
   public void add (Component c)     {}  // también puede lanzar una excepción
   public void remove (Component c)  {}  // también puede lanzar una excepción
   public Component getChild (int i) { return null; }
}

public class Leaf extends Component {
}

public class Composite extends Component {
   private List<Component> children = new ArrayList<Component>();
   public void add (Component c)     { children.add (c); }
   public void remove (Component c)  { children.remove (c); }
   public Component getChild (int i) { return children.get(i); }
}
```

Composite

Código Ejemplo.

```
public interface Component {
   public Composite getComposite ();
}
public class Leaf implements Component {
   public Composite getComposite () { return null; }
}
public class Composite implements Component {
   private List<Component> children = new ArrayList<Component>();
   public void add (Component c)     { children.add(c); }
   public void remove (Component c)  { children.remove (c); }
   public Component getChild (int i) { return children.get(i); }
   public Composite getComposite ()  { return this; }
}

// código cliente
Composite aComposite = new Composite();
Leaf aLeaf = new Leaf();
Component aComponent = aComposite;
if (aComponent.getComposite()!=null)
   aComponent.add(new Leaf());  // añadirá la hoja
aComponent = aLeaf;
if (aComponent.getComposite()!=null)
   aComponent.add(new Leaf());  // no añadirá la hoja
```

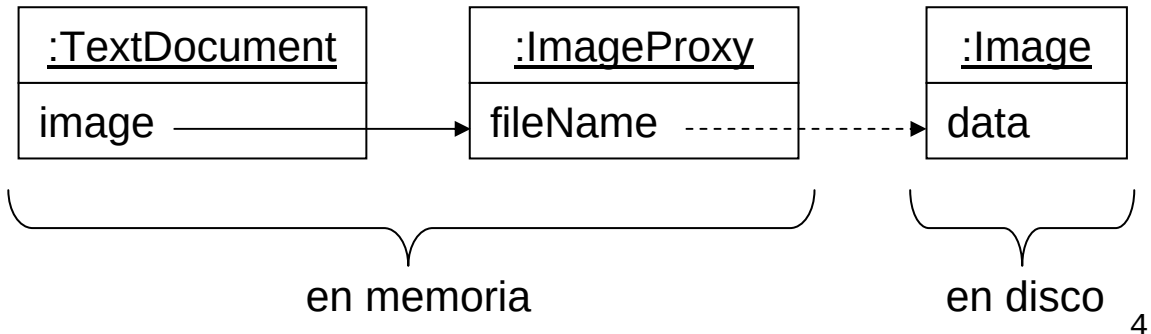
Composite

En Java...

- En el paquete java.awt.swing
  - **Component**
    - Component
  - **Composite**
    - Container (abstracta)
    - Panel (concreta)
    - Frame (concreta)
    - Dialog (concreta)
  - **Leaf**:
    - Label
    - TextField
    - Button

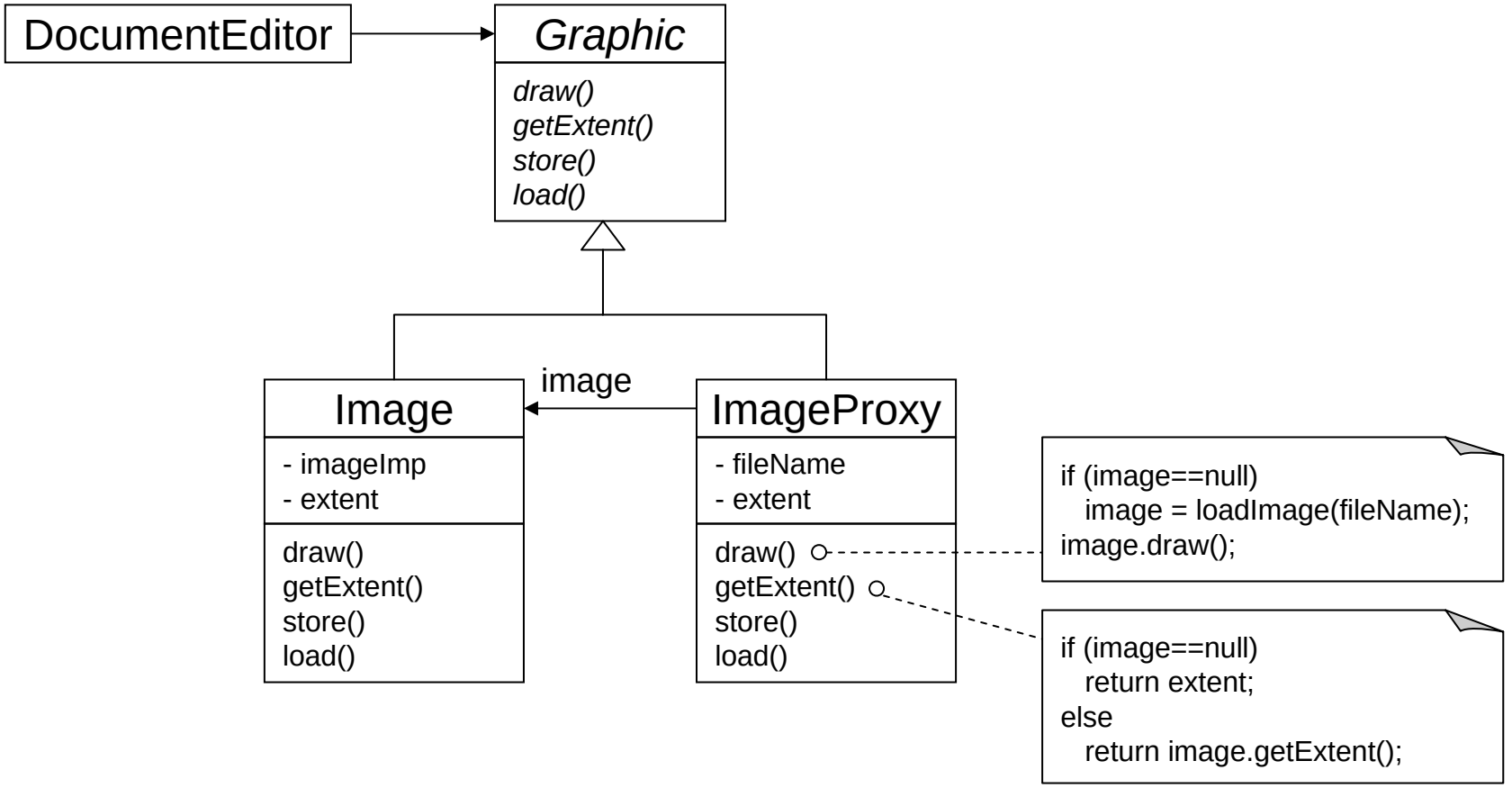
Proxy

- **Propósito:**
  - Proporcionar un representante o sustituto de otro objeto para controlar el acceso a éste
- **Motivación**
  - Retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario. Por ejemplo, no abrir las imágenes de un documento hasta que no son visibles.
- **Solución:**



Proxy

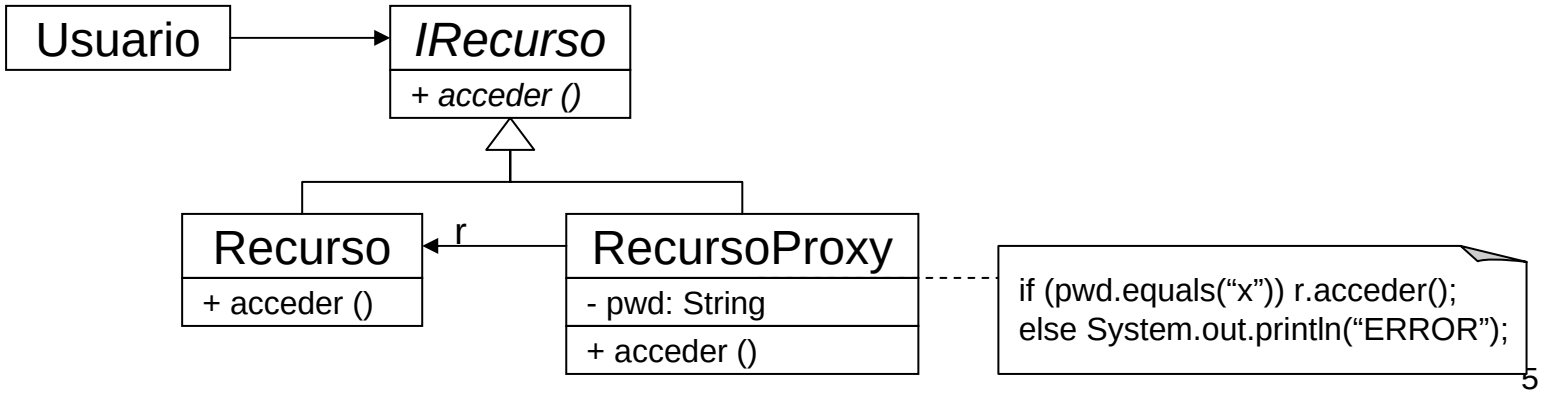
Motivación



Proxy

Aplicabilidad

- El patrón *Proxy* se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero. Por ejemplo:
  - **Proxy virtual**: crea objetos costosos por encargo (como la clase *ImageProxy* en el ejemplo de motivación)
  - **Proxy remoto**: representa un objeto en otro espacio de direcciones
  - **Referencia inteligente**: sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto (ej. contar número de referencias, cargar un objeto persistente en memoria, bloquear el objeto para impedir acceso concurrente, ...)
  - **Proxy de protección**: controla el acceso a un objeto



Proxy

Estructura

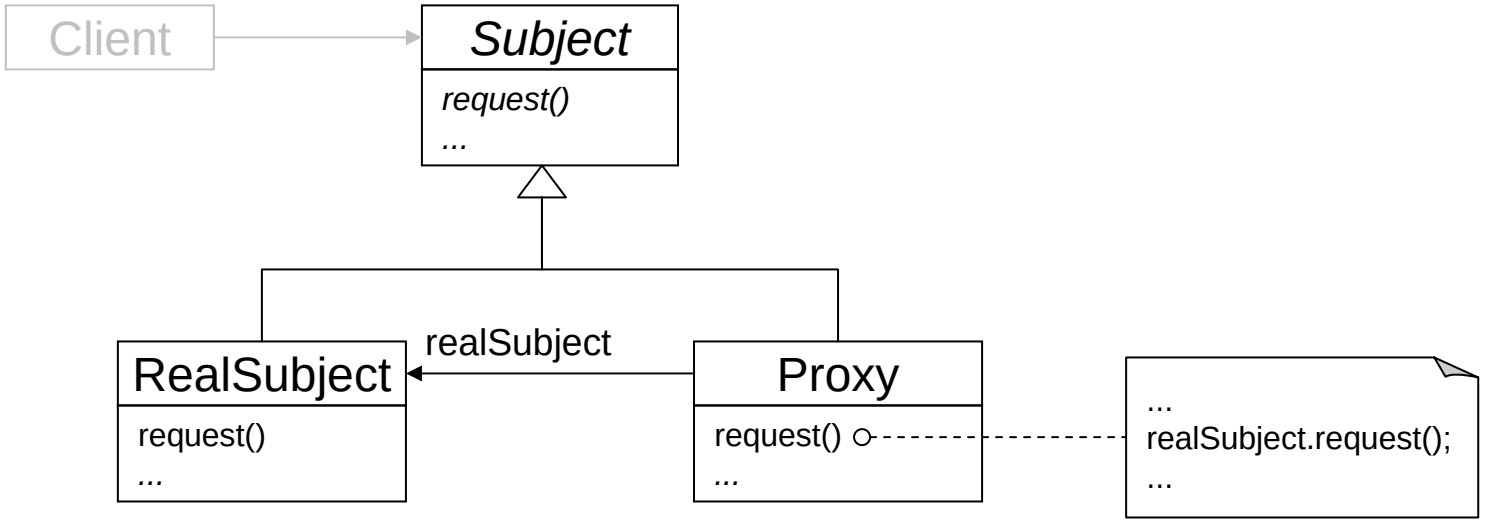
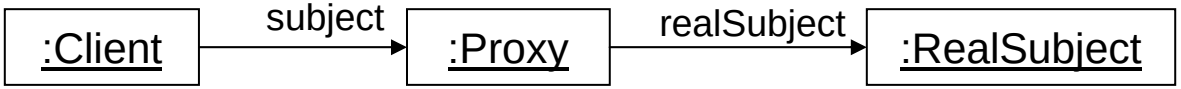


Diagrama de objetos



Proxy

Participantes

- **Proxy** (*ImageProxy*):
  - Mantiene una referencia al objeto real
  - Proporciona una interfaz idéntica a la del objeto real
  - Controla el acceso al objeto real, y puede ser responsable de crearlo y borrarlo
  - Otras responsabilidades dependen del tipo de proxy:
    - Proxies remotos: codifican las peticiones, y las envían al objeto real
    - Proxies virtuales: pueden guardar información del objeto real (caché)
    - Proxies de protección: comprueban que el cliente tiene los permisos necesarios para realizar la petición
- **Subject** (*Graphic*): define una interfaz común para el proxy y el objeto real, de tal modo que se puedan usar de manera indistinta
- **RealSubject** (*Image*): clase del objeto real que el proxy representa

Proxy

Consecuencias

- Introduce un nivel de indirección con diferentes usos:
  - Un proxy remoto puede ocultar el hecho de que un objeto reside en otro espacio de direcciones
  - Un proxy virtual puede realizar optimizaciones, como la creación de objetos bajo demanda
  - Los proxies de protección y las referencias inteligentes permiten realizar tareas de mantenimiento adicionales al acceder a un objeto

- Optimización copy-on-write
  - Copiar un objeto grande puede ser costoso
  - Si la copia no se modifica, no es necesario incurrir en dicho gasto
  - El sujeto mantiene un número de referencias, y sólo cuando se realiza una operación que modifica el objeto, éste se copia

Proxy

Implementación (Proxy Virtual)

```
public abstract class Graphic {
   public void draw();
}


public class Image extends Graphic {
   public void draw() { ... }
}


public class ImageProxy extends Graphic {
   private Image  image;
   private String fileName;
   public ImageProxy (String fileName) {
      this.fileName = fileName;
      this.image    = null;
   }
   public Image loadImage() { ... }
   public void draw () {
      if (image==null)
         image = loadImage(fileName);
      image.draw();
   }
}
```

```
public class TextDocument {
   public void insert (Graphic g) {
    ...
   }
}


// código para insertar una ImageProxy
// en un documento
TextDocument td = new TextDocument();
Graphic g =
   new ImageProxy("imagen.gif");
td.insertar(g);
```

Indice

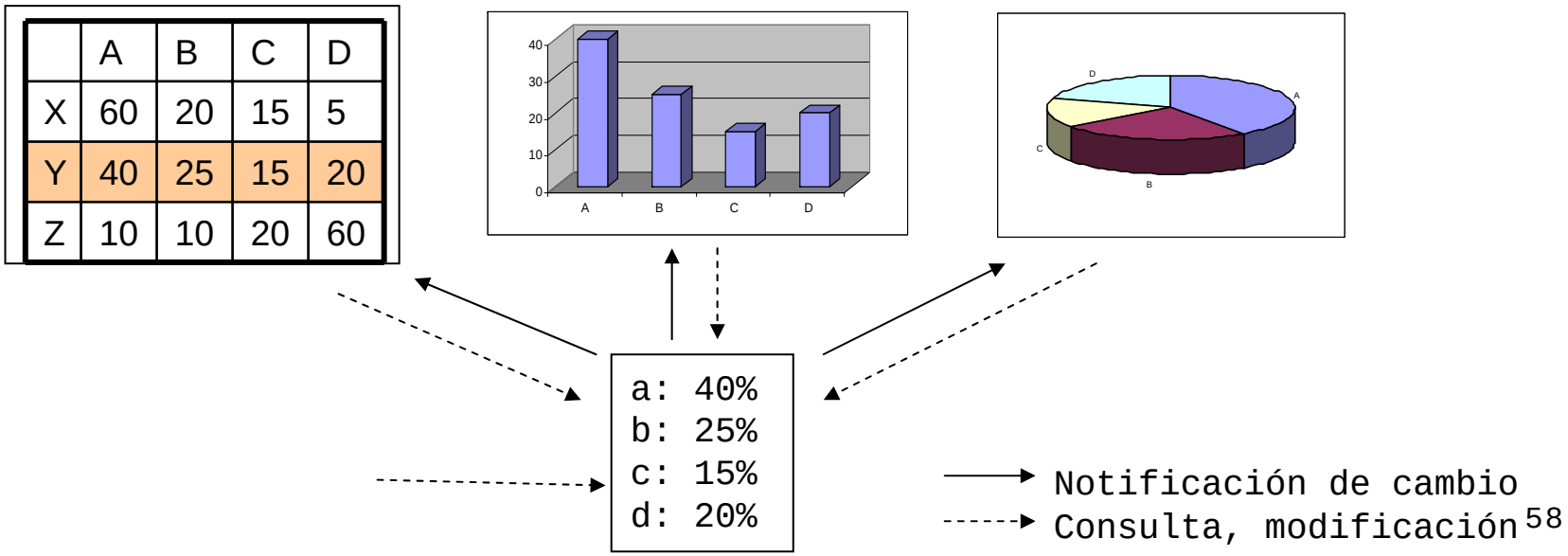
- Introducción.
- Patrones de Creación.
- Patrones Estructurales.
- **Patrones de Comportamiento.**
  - **Observer.**
  - **Iterator.**
- Conclusiones.
- Bibliografía.

Observer

- **Propósito:**
  - Define una dependencia de uno-a-muchos entre objetos de forma que, cuando un objeto cambia de estado, se notifica a los objetos dependientes para que se actualicen automáticamente.
  - También conocido como *dependents*, *publish-subscribe*

Observer

- **Motivación:**
  - Mantener la consistencia entre objetos relacionados, sin aumentar el acoplamiento entre clases
  - Ej: separación de la capa de presentación en una interfaz de usuario de los datos de aplicación subyacentes



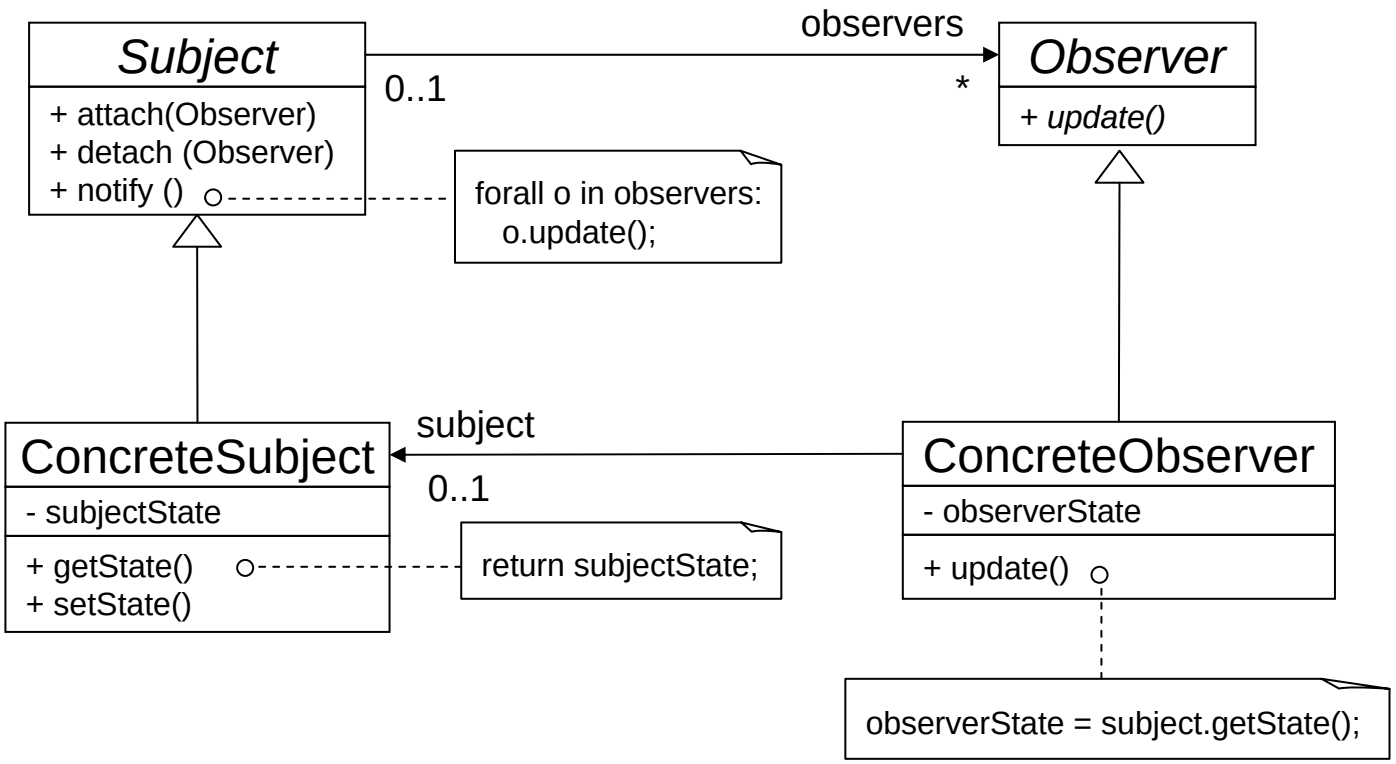
Observer

Aplicabilidad

- Usa el patrón *Observer*:
  - Cuando una abstracción tiene dos aspectos, y uno depende del otro. Encapsular los aspectos en objetos distintos permite cambiarlos y reutilizarlos.
  - Cuando cambiar un objeto implica cambiar otros, pero no sabemos exactamente cuántos hay que cambiar
  - Cuando un objeto debe ser capaz de notificar algo a otros sin hacer suposiciones sobre quiénes son dichos objetos. Esto es, cuando se quiere bajo acoplamiento.

Observer

Estructura



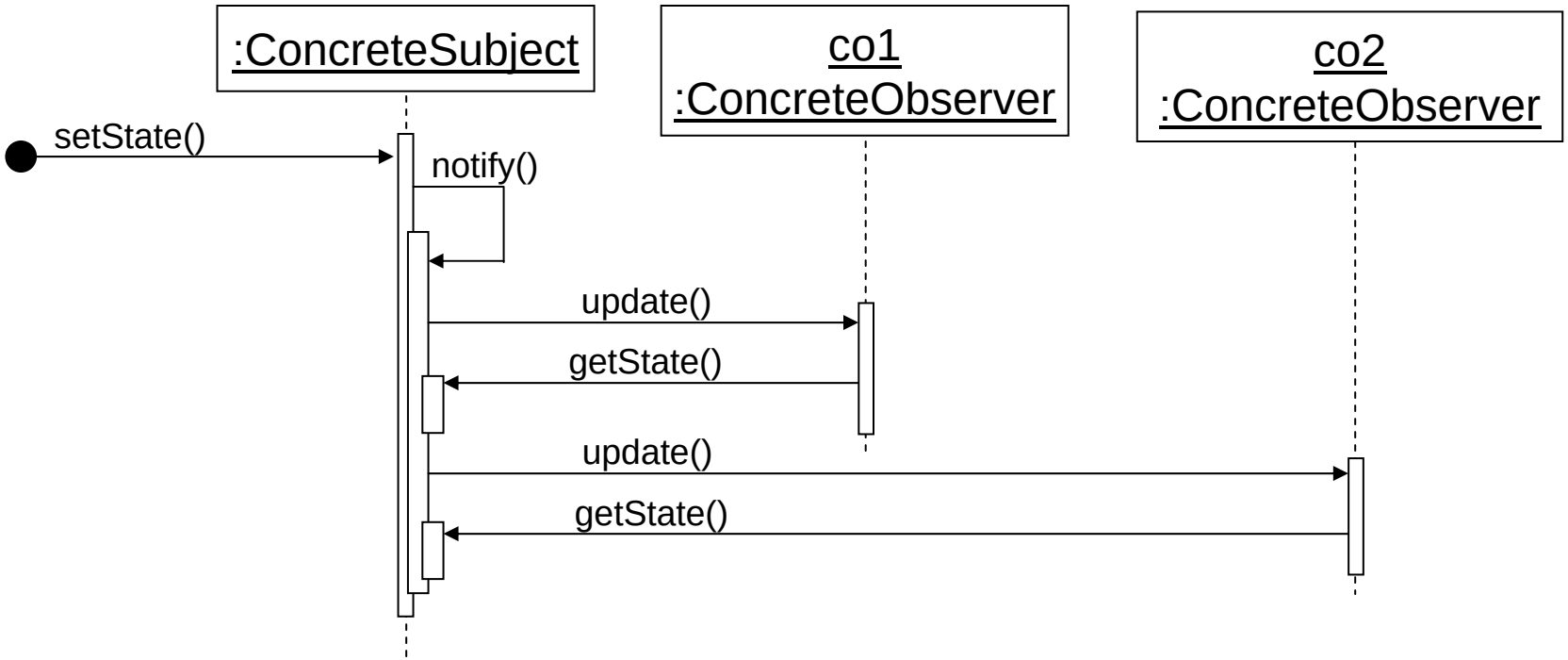
Observer

Participantes

- **Subject**:
  - conoce a sus observadores, que pueden ser un número arbitrario
  - proporciona una interfaz para añadir y quitar objetos observadores
- **Observer**:
  - define la interfaz de los objetos a los que se deben notificar cambios en un sujeto
- **ConcreteSubject***:*
  - almacena el estado de interés para sus observadores
  - envía notificaciones a sus observadores cuando su estado cambia
- **ConcreteObserver**:
  - mantiene una referencia a un *ConcreteSubject*
  - almacena el estado del sujeto que le resulta de interés
  - implementa la interfaz de *Observer* para mantener su estado consistente con el del sujeto

Observer

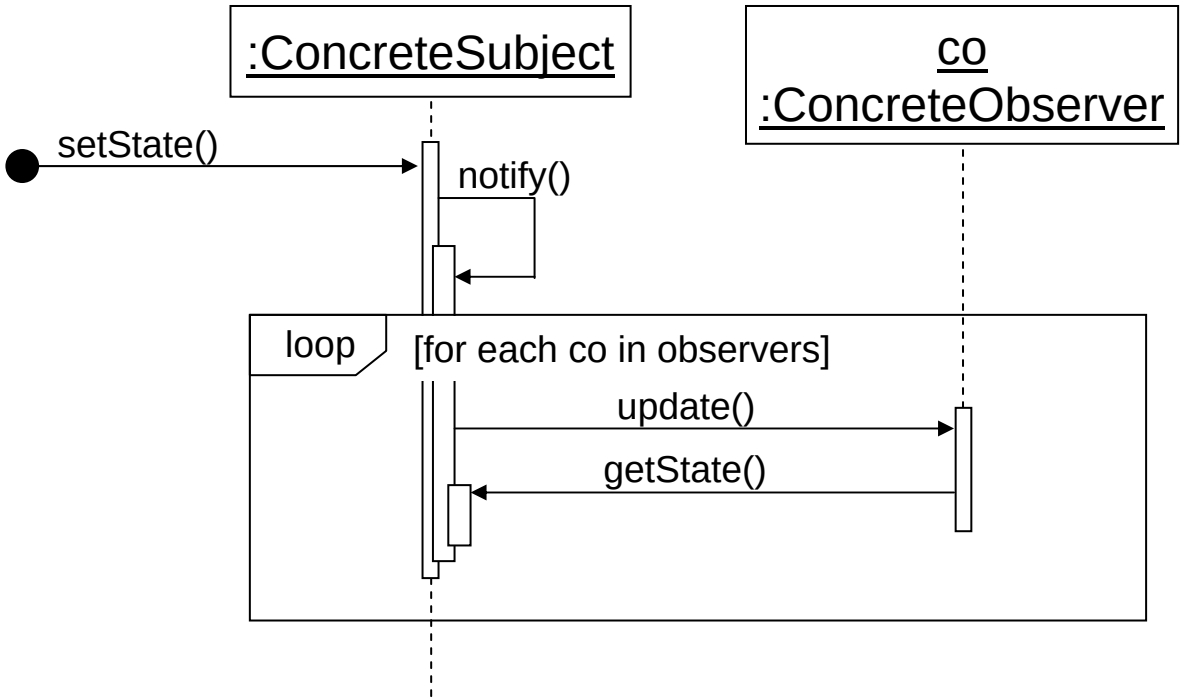
Colaboraciones



(sujeto con dos observadores)

Observer

Colaboraciones



(sujeto con un número arbitrario de observadores)

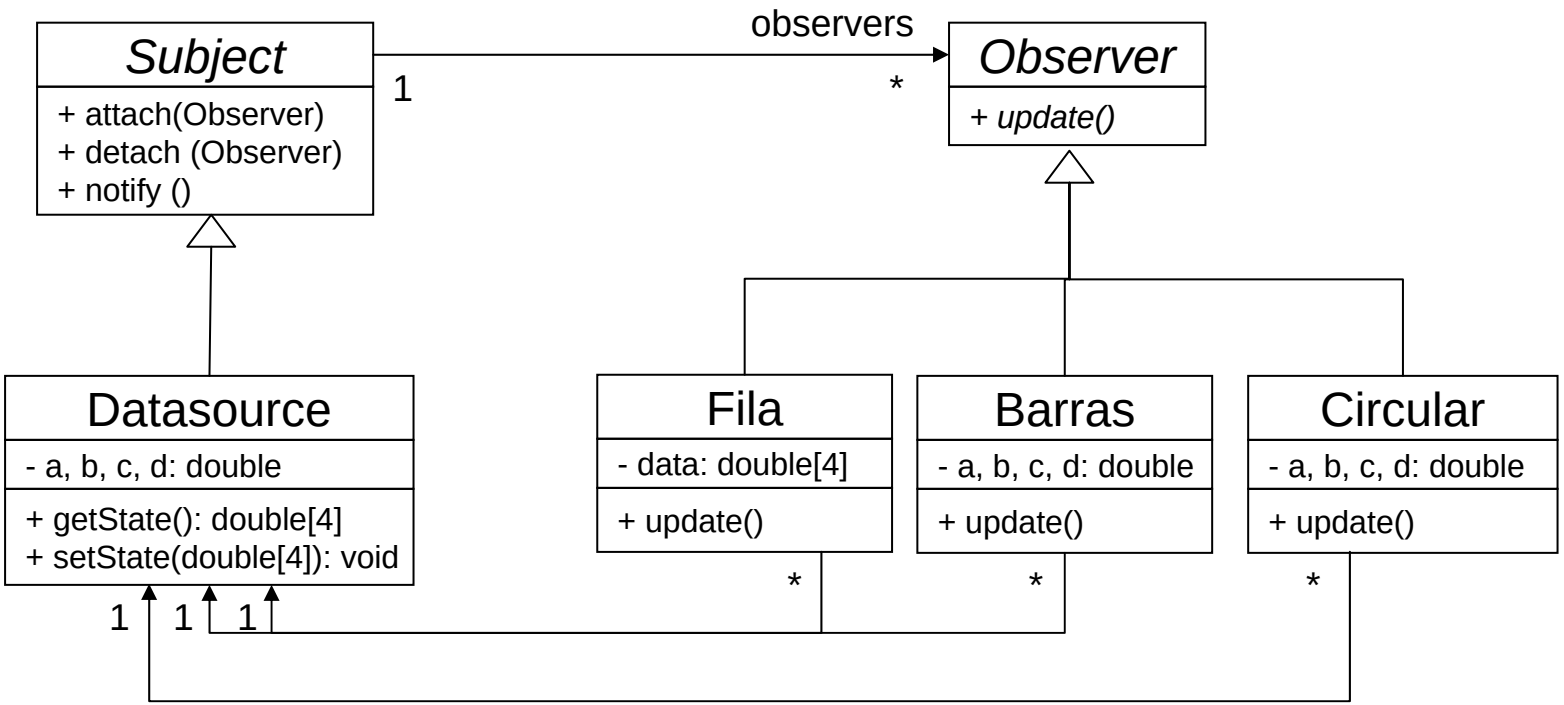
Observer

Consecuencias

- Permite modificar sujetos y observadores de manera independiente
- Permite reutilizar un sujeto sin reutilizar sus observadores, y viceversa
- Permite añadir observadores sin tener que cambiar el sujeto ni los demás observadores
- Acoplamiento abstracto entre el sujeto y el observador. El sujeto no sabe la clase concreta de sus observadores (acoplamiento mínimo).
- Soporte para *broadcast*. El sujeto envía la notificación a todos los observadores suscritos. Se pueden añadir/quitar observadores.
- Actualizaciones inesperadas. Una operación en el sujeto puede desencadenar una cascada de cambios en sus observadores. El protocolo no ofrece detalles sobre lo que ha cambiado.

Observer

Ejemplo



Observer

Código del ejemplo

```
public abstract class Subject {
 protected List<Observer> observers;

 public Subject() {
    observers =
      new ArrayList<Observer>();
 }
 public void attach(Observer o) {
    observers.add(o);
 }
 public void detach(Observer o) {
    observers.remove(o);
 }
 public void notify() {
    Iterator<Observer> it;
    it = observers.iterator();
    while (it.hasNext())
       it.next().update();
 }
}
```

```
public class Datasource
             extends Subject {
  private double a, b, c, d;
  public double[] getState ()
  {
    double[] data = new double[4];
    data[0] = a;
    data[1] = b;
    data[2] = c;
    data[3] = d;
    return data;
  }
  public void setState(double[] dd)
  {
    a = dd[0];
    b = dd[1];
    c = dd[2];
    d = dd[3];
    this.notify();
  }
}
```

Observer

Código del ejemplo

```
public abstract class Observer {
  protected Subject subject;

  public Observer (Subject s) {
    subject = s;
    subject.attach(this);
 }

  public abstract void update();
}
```

```
public class Fila extends Observer {
  private double[] data;

  public Fila (Subject s) {
    super(s);
    data = new double[4];
  }
  public void update () {
    double[4] data;
    data = ((Datasource)subject).
              getState();
    for (int i=0; i<4; i++)
       this.data[i] = data[i];
    this.redraw();
  }

  public void redraw () { ... }
}
```

Observer

en Java…

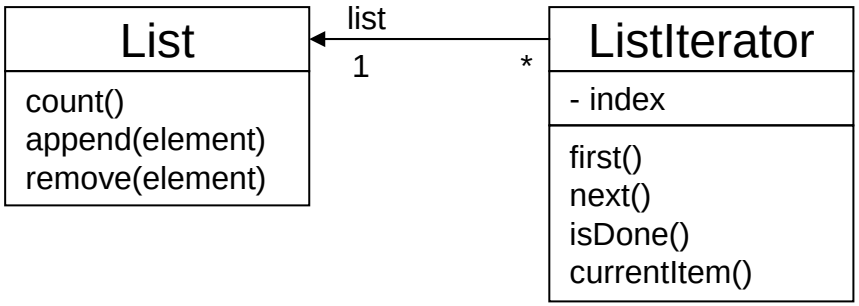
- La interfaz `java.util.Observer`
  - `void update (Observable o, Object arg)`
- La clase `java.util.Observable`
  - `Observable()`
  - `void addObserver(Observer)`
  - `int countObservers()`
  - `void deleteObserver(Observer o)`
  - `void deleteObservers()`
  - `void notifyObservers()`
  - `void notifyObservers(Object arg)`
  - `boolean hasChanged()`
  - `void clearChanged()`
  - `void setChanged()`

Iterator

- **Propósito:**
  - Proporcionar acceso secuencial a los elementos de un agregado, sin exponer su representación interna.
  - También conocido como cursor
- **Motivación:**
  - Ej: Una lista debe proporcionar un medio de navegar por sus datos sin exponer su estructura interna
  - Se debe poder atravesar la lista de varias maneras, pero no añadir operaciones a la lista por cada tipo de recorrido
  - Se debe poder realizar varios recorridos simultáneamente
- **Solución:**
  - Dar la responsabilidad de recorrer la lista a un objeto iterador

Iterator

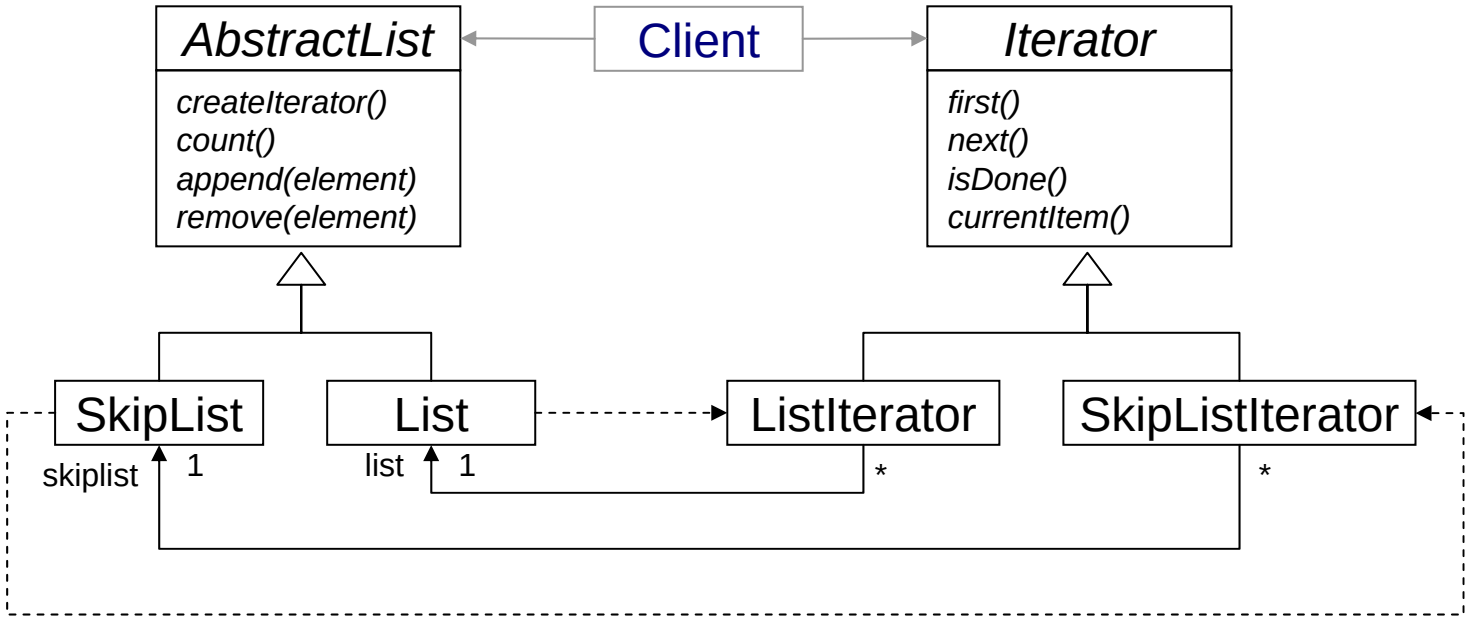
Motivación



- Al instanciar *ListIterator* se debe proporcionar la lista
- Una vez instanciado el iterador, se puede acceder a los elementos de la lista
- Ventaja: separar el mecanismo de recorrido del objeto lista permite definir iteradores que implementen distintas estrategias, varios recorridos a la vez
- Inconvenientes:
  - No se asegura una interfaz común para todos los iteradores de lista
  - Iterador y cliente están acoplados, ¿cómo sabemos qué iterador usar?
  - No se asegura una interfaz común para la creación de listas

Iterator

Motivación



- Generalizar el iterador para que soporte iteración polimórfica
- Se puede cambiar el agregado sin cambiar el código cliente
- Las listas se hacen responsables de crear sus propios iteradores

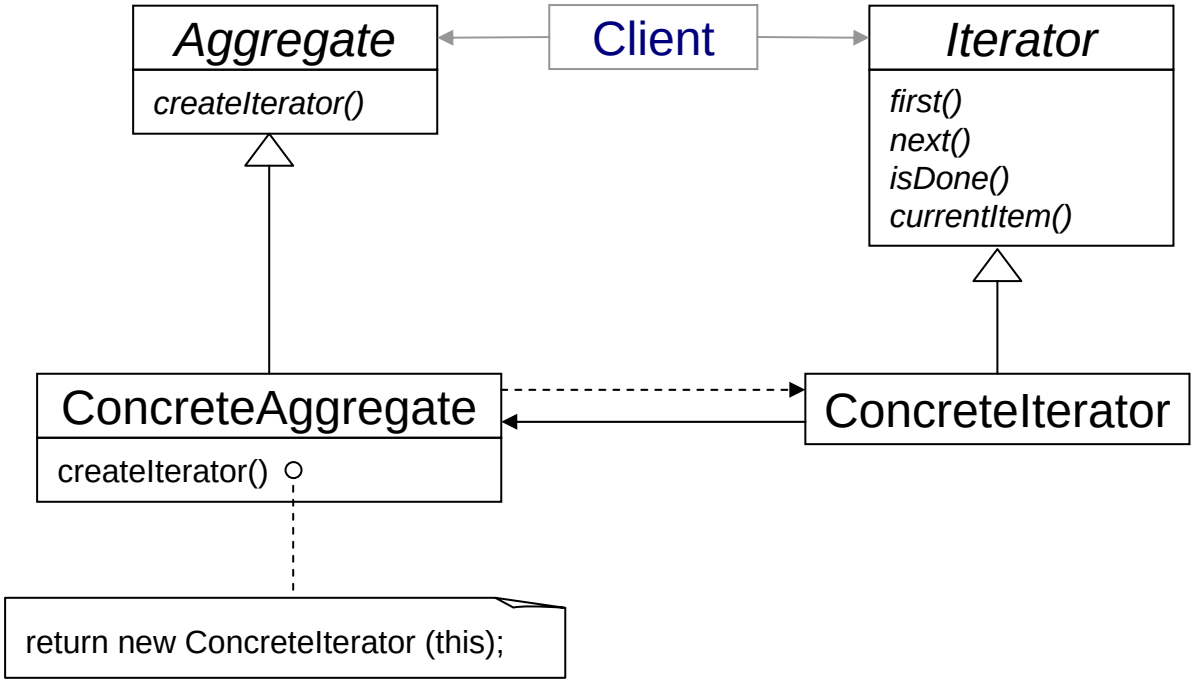
Iterator

Aplicabilidad

- Usa el patrón *Iterator*:
  - Para acceder al contenido de un agregado sin exponer su representación interna
  - Para permitir varios recorridos sobre un agregado
  - Para proporcionar una interfaz uniforme para recorrer distintos tipos de agregados (esto es, permitir iteración polimórfica)

Iterator

Estructura



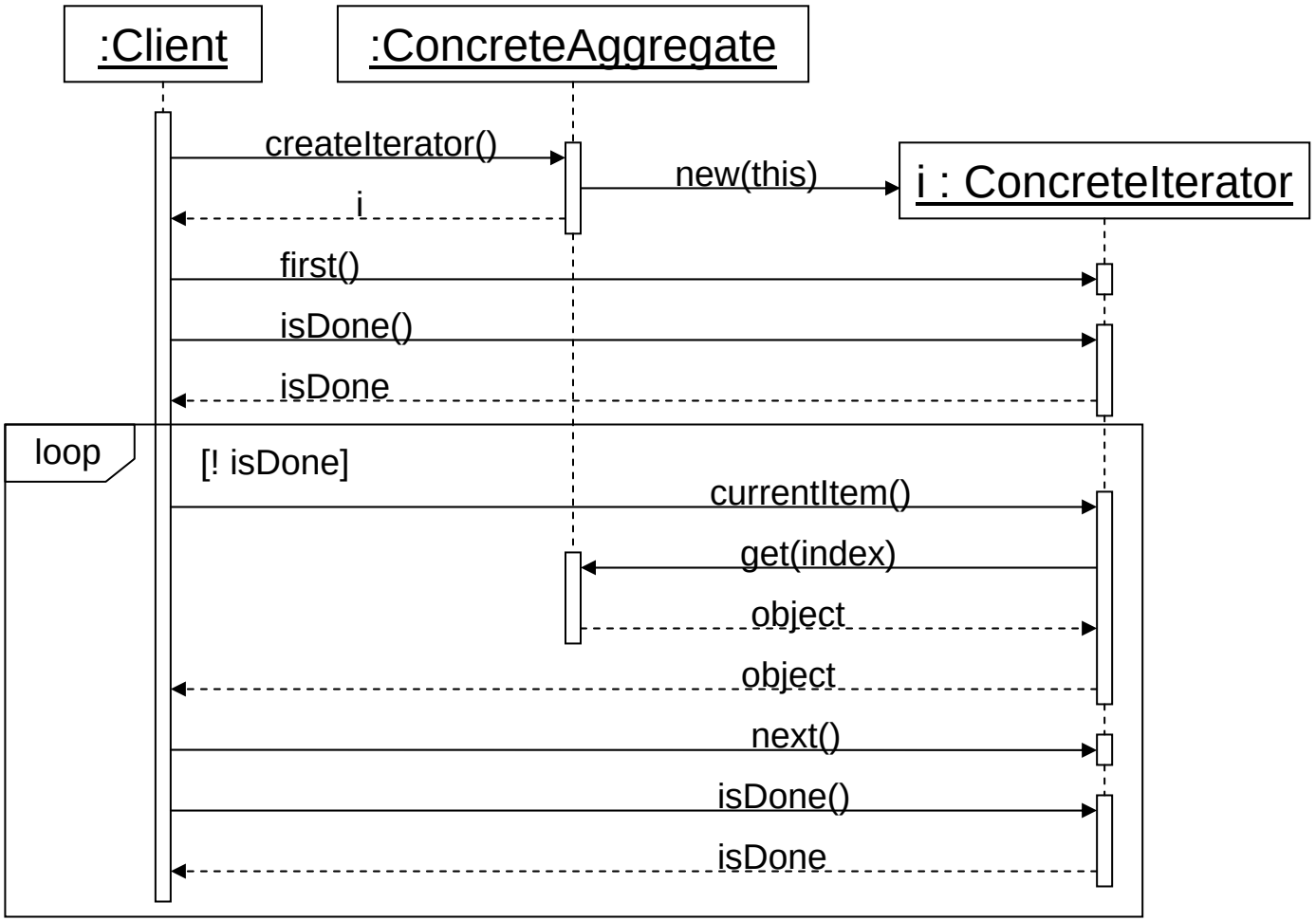
Iterator

Participantes

- **Iterator**: define interfaz para acceder y recorrer elementos
- **ConcreteIterator**:
  - Implementa la interfaz *Iterator*
  - Mantiene la posición actual en el recorrido del agregado
- **Aggregate***:* define interfaz para crear un objeto *Iterator*
- **ConcreteAggregate***:* implementa una interfaz de creación del *Iterator* para devolver la instancia de *ConcreteIterator* apropiada

Iterator

Colaboraciones



Iterator

Código del Ejemplo

```
public interface Iterator {
   public void first();
   public void next();
   public boolean isDone();
   public Object currentItem();
}

public interface Aggregate {
   public Iterator createIterator();
   public Object get(int);
   public int count();
}

public class ListIterator {
   private Aggregate a;
   private int current;
   public ListIterator (Aggregate a); {
      this.a = a;
      current = 0;
   }
   public void first() { current = 0; }
   public void next() { current++; }
   public boolean isDone() { return current >= a.count(); }
   public Object currentItem() { return a.get(current); }
}
```

Iterator

Implementación

- ¿Quién controla la iteración?
  - El cliente: iterador externo
    - Más flexible, permite comparar dos colecciones

```
Iterator it = list.createIterator();
it.first();
while (it.isDone() == false) {
   it.next();
   it.currentItem();
}
```

  - El iterador: iterador interno
    - El iterador recibe una función a aplicar sobre los elementos del agregado, y el recorrido es automático
    - Simplifica el código del cliente

```
Iterator it = list.createIterator(OPERATION);
it.traverse();
```

Iterator

En Java…

- Marco de contenedores de java (*framework collection*)
  - **Aggregate**:
    - Las interfaces `Collection`, `Set`, `SortedSet`, `List`, `Queue` de `java.util`
    - Incluyen método `iterator()`, que devuelve un iterador genérico
  - **ConcreteAggregate**: implementaciones de esas interfaces
    - Set es implementada por las clases `HashTree`, `TreeSet`, `LinkedHashSet`
    - List es implementada por las clases `ArrayList`, `LinkedList`
  - **Iterator**: interfaz `java.util.Iterator`
    - `boolean hasNext()`
    - `Object next()`
    - `void remove()`
  - **ConcreteIterator**: implementaciones concretas de `Iterator`
  - **Ejemplo de cliente**:

```
java.util.Collection c = new java.util.LinkedList();
java.util.Iterator it = c.iterator();
while (it.hasNext()) {
        it.next();
}
```

Resumen

- Orientación a Objetos: Aplicación como conjunto de objetos que interactúan.
- Conceptos:
  - Clases, Objetos, Encapsulación, Polimorfismo y Herencia.
- Ventajas:
  - Extensibilidad, reutilización.
  - Modela el mundo real de manera natural.

Conclusiones

- Los patrones de diseño describen la solución a problemas que se repiten una y otra vez en nuestros sistemas, de forma que se puede usar esa solución siempre que haga falta
- Capturan el conocimiento que tienen los expertos a la hora de diseñar
- Ayudan a generar software “maleable” (software que soporta y facilita el cambio, la reutilización y la mejora)
- Son guías de diseño, no reglas rigurosas

Bibliografía

- “UML distilled: a brief guide to the standard Object Modelling Language”, Martin Fowler. Editorial Addison-Wesley.
- “Design patterns: elements of reusable object oriented software”, Erich Gamma et al. Editorial Addison- Wesley.
- “Patterns in Java: a catalog of reusable design patterns illustrated with UML”, Mark Grand. Editorial John Wiley & sons.