Diseño: Patrones de Diseño
[Gamma et al. 1995]

- Los patrones de diseño describen un problema que ocurre una y otra vez y entrega una buena solución.
- Descripción:
  - Nombre
  - Problema (cuando usarlo)
  - Solución (diagrama de clases, de interacción, código)
  - Consecuencias (ventajas y desventajas)
- Ejemplos: iterator, adapter, template method, abstract factory y observer, entre otros.
- Nota: algunos *diagramas usados para describir los patrones no siguen la notación UML pero es parecida. El código está en c++.*

Diseño: Patrones de Diseño

Ventajas:

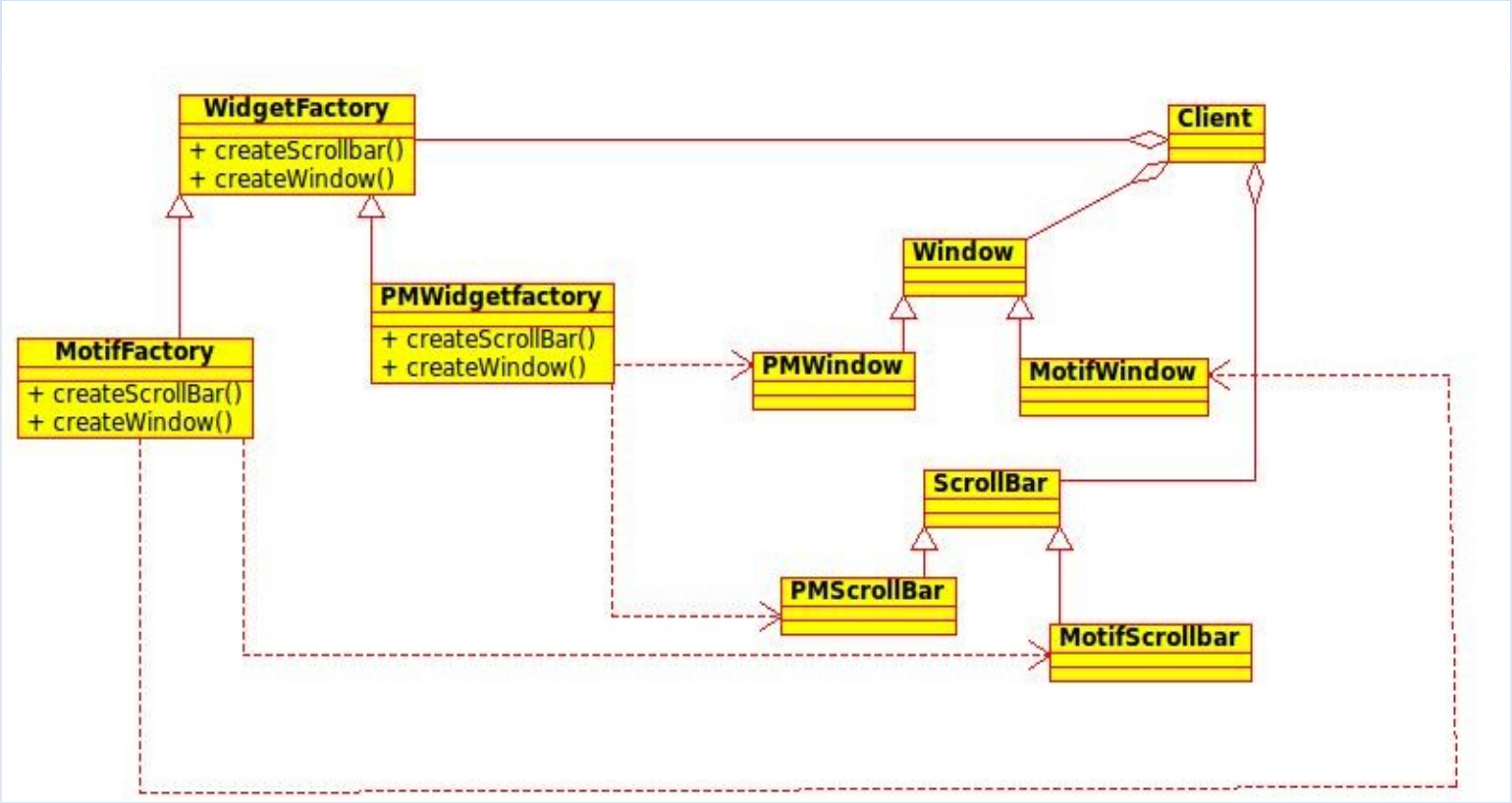
- Reuso de diseños exitosos => nuevos diseños correctos más rápidamente.
- Desarrollo de sistemas reusables
- Simplifica la documentación

Abstract Factory

- **Objetivo:** proveer una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
- **Motivación:** Portabilidad de una Toolkit a distintos ambientes, por ejemplo a Motif y Presentation Manager. Para poder ser portables a look-and-feels estándares, no debemos escribir la aplicación usando directamente las instrucciones de sus widgets (scroll bars, windows, y buttons), sino definir una clase abstracta WidgetFactory que declara una interfaz para cada tipo básico de widget. Las subclases concretas implementan los widgets para un look-and-feel específico.

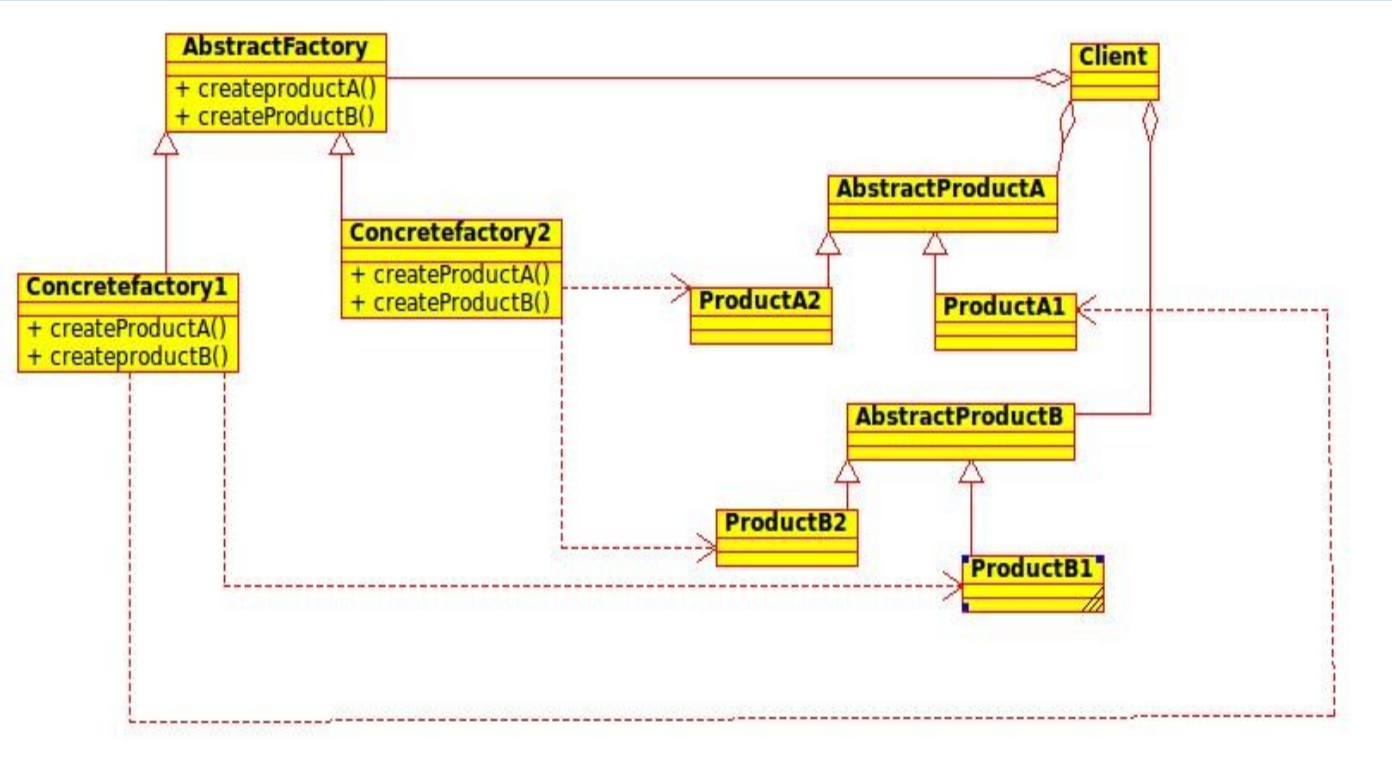
Abstract Factory

Ejemplo:



Abstract Factory

Estructura:



Abstract factory

Ejemplo: Construcción de laberintos

Usaremos la fábrica abstracta Mazefactory para construir laberintos. Mazefactory permite crear sus partes tales como piezas, paredes y puertas entre piezas, etc.

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

       virtual Maze* makeMaze()
           { return new Maze; }
       virtual Wall* makeWall()
           { return new Wall; }
       virtual Room* makeRoom(int n)
           { return new Room(n); }
       virtual Door* makeDoor(Room* r1, Room* r2)
           { return new Door(r1, r2); }
   };
```

Abstract Factory: Laberinto

El siguiente método de MazeGame crea un laberinto de dos piezas y una puerta. Recibe la fábrica a usar como parámetro:

```
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;
}
```

Abstract Factory: Laberinto

Creación de fábricas concretas:

```
    class EnchantedMazeFactory : public MazeFactory {
    public:
        EnchantedMazeFactory();
        virtual Room* makeRoom(int n)
            { return new EnchantedRoom(n, castSpell()); }

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

    ...
    };
class BombedMazeFactory : public MazeFactory {
    public:
        BombedMazeFactory();
        virtual Room* makeRoom(int n)
            { return new RoomWithABomb(n); }

        virtual Wall* makeWall()
            { return new BombedWall(); }

    ...
    };
```

AbstractFactory: Laberinto

```
class Wall  {
    public:
        Wall();

    ...
    };
```

```
class Room  {
    public:
        Room(int n);

    ...
    };
```

```
class RoomWithABomb: public Room  {
    public:
        RoomWithABomb(int n);

    ...
    };
```

```
class EnchantedRoom: public Room  {
    public:
      EnchantedRoom(int n, ....);

    ...
    };
```

Abstract Factory: Laberinto

Uso de una fabrica concreta:

```
MazeGame game;

BombedMazeFactory bombedfactory;
game.createMaze(bombedfactory);

MazeGame newgame;
EnchantedMezaFactory  enchantedfactory;
newgame.createMaze(enchantedfactory);
```

Abstract factory

- **Aplicabilidad:**
  - Sistema independiente de cómo los productos son creados, compuestos y representados.
  - Sistemas es configurado con una de las familias de productos
  - Una familia de productos relacionados está diseñada para ser usada en conjunto.
  - Si se desea proveer una biblioteca de productos, sólo se revela la interfaz
- **Resultados**
  - La fábrica encapsula la responsabilidad y el proceso de crear objetos productos, por lo tanto aisla al cliente de la creación de estos.
  - La fábrica permite el intercambio de familias productos fácilmente.
  - Incentiva la consistencia entre productos.
  - Incluir nuevos tipos de productos requiere la modificación de clases existentes

Adapter

- **Adapter:** Cambia la interfaz de una clase en otra que el cliente espera

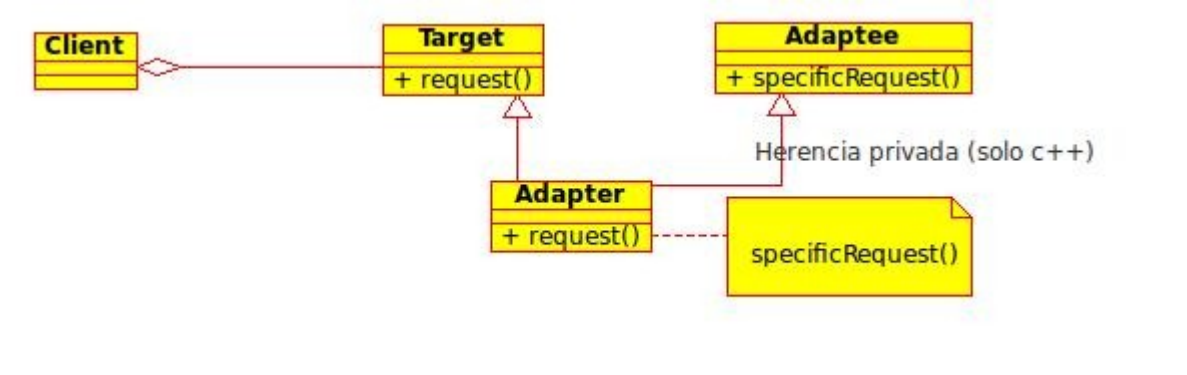
  Motivación: Un editor de dibujos usa elementos gráficos tales como líneas, texto, círculos, entre otros, para crear cajas o cuadros. Su clase gráfica abstracta se llama Shape y de esta derivan TextShape, LineShape y PolygonShape. En el sistema existe textView, una clase para crear y editar texto. La interfaz de TextView es distinta a la que queremos para Shape.

Adapter

- ¿Cómo modelar TextShape?
- Adapter analiza dos alternativas (una correcta y la otra incorrecta):
  - Definir TextShape como subclase de Shape y de TextView pero heredando de ésta última en forma privada
  - Definir TextShape como subclase de Shape e incluir TextView como variable de intancia de TextShape (agregación) e implementar la interfaz de TextShape en términos de la interfaz de TextView

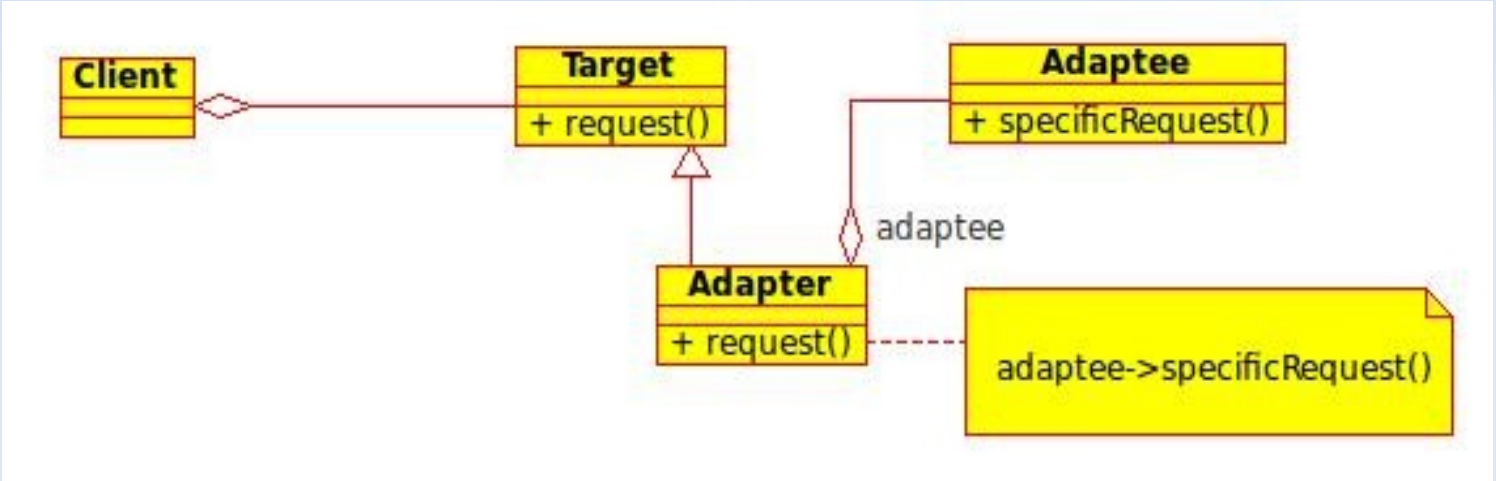
Adapter con herencia privada

Estructura:



Adapter usando agregación

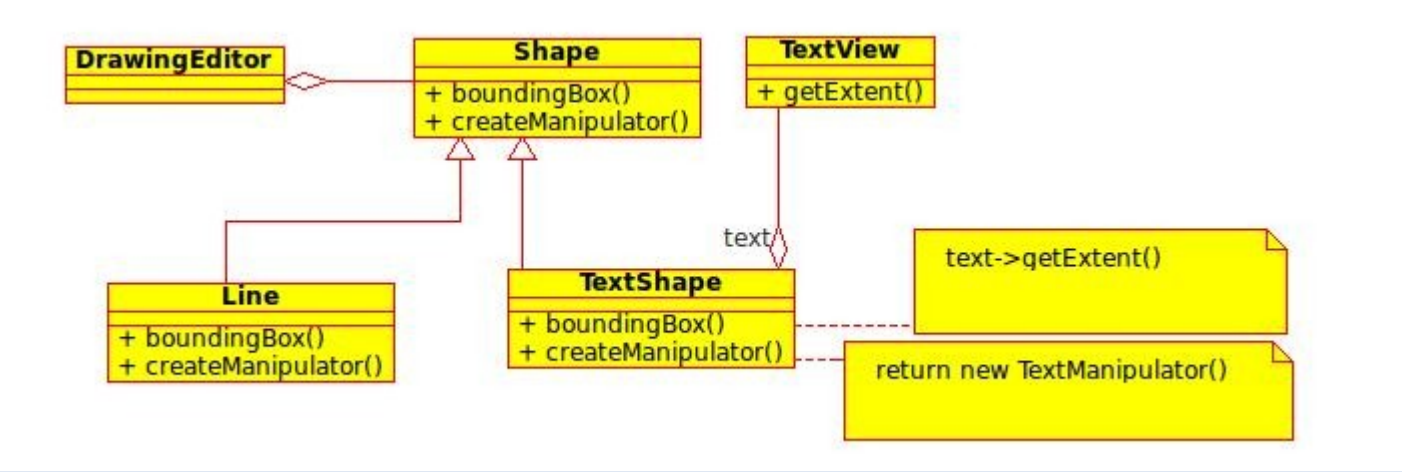
Estructura:



Nota: "adaptee" es una variable de instancia de Adapter; contiene un puntero a un objeto de tipo Adaptee

Adapter: Ejemplo

Usando agregación:



Adapter: Código del ejemplo concreto usando herencia privada

```
class TextShape : public Shape, private TextView {
    public:
        TextShape();
        virtual void boundingBox(Point& bottomLeft, Point& topRight) ;
        virtual bool isEmpty();
        ...
}
```

Adapter: Código del ejemplo concreto usando herencia privada

```
void TextShape::BoundingBox (Point& bottomLeft, Point& topRight ) {
        Coord bottom, left, width, height;
        getOrigin(bottom, left);
        getExtent(width, height);
        bottomLeft.set(left,bottom);
        topRight.set(left + width, bottom + height);
}
bool TextShape::isEmpty () {
        return TextView::isEmpty();
}
```

Adapter: Código del ejemplo usando agregación

```
class TextShape : public Shape {
    public:
        TextShape(TextView*);
        virtual void boundingBox(Point& bottomLeft, Point& topRight);
        virtual bool isEmpty();
        ...
    private:
        TextView* _text;
};
```

Adapter: Código del ejemplo concreto usando agregación

```
TextShape::TextShape (TextView* t) {
        _text = t;
}
void TextShape::boundingBox(Point& bottomLeft, Point& topRight ) {
        Coord bottom, left, width, height;
        _text->getOrigin(bottom, left);
        _text->getExtent(width, height);
        bottomLeft.set(left, bottom);
        topRight.set(left + width, bottom + height);
}
bool TextShape::isEmpty () {
        return _text->isEmpty();
}
```

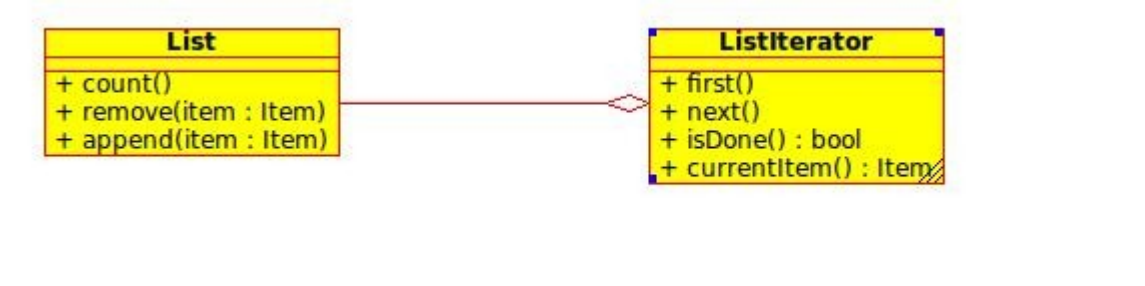
Adapter

- ¿Cuál alternativa usar y por qué?
  - Alternativa con agregación (APROPIADA)
    - Satisface la relación de subtipos
    - Es más flexible, pues permite implementar la nueva interfaz con objetos de TextView o con descendientes de él.
  - Alternativa con herencia privada (NO APROPIADA)

- El uso de herencia privada muestra directamente que no hay una relación de subtipos entre TextView y TextShape (interfaces son distintas).
- Lo anterior impide usar TextShape en el contexto de TextView. ¿Para qué usar herencia si una clase no se va a usar en el contexto de la otra?
- No permite usar un descendiente de TextView en lugar de TextView sin cambiar la clase TextShape

Iterator

Provee acceso a un objeto referenciado (agregación) sin mostrar su implementación



Iterator

```
template <class Item>
        class List {
        public:
            List(long size = DEFAULT_LIST_CAPACITY);

            long count();
            Item& get(long index);
            // ...
        };

    template <class Item>
        class Iterator {
        public:
            virtual void first() = 0;
            virtual void next() = 0;
            virtual bool isDone() = 0;
            virtual Item currentItem() = 0;
        protected:
            Iterator();
        };
```

Iterator: Implementación

```
template <class Item>
        class ListIterator : public Iterator<Item> {
        public:
            ListIterator(const List<Item>* aList);
            virtual void first();
            virtual void next();
            virtual bool isDone();
            virtual Item currentItem();

        private:
            const List<Item>* _list;
            long _current;
        };
```

```
template <class Item>
ListIterator<Item>::ListIterator(const List<Item>*aList):
                                  _list(aList), _current(0){}
```

Iterator: Implementación

```
template <class Item>
void ListIterator<Item>::first () {
     _current = 0;
}

template <class Item>
void ListIterator<Item>::next () {
     _current++;
}

template <class Item>
bool ListIterator<Item>::isDone ()  {
     return _current >= _list->count();
}

template <class Item>
Item ListIterator<Item>::currentItem ()  {
     if (isDone()) {
        throw IteratorOutOfBounds;
     }
     return _list->get(_current);
}
```

Uso de iteradores:

```
 void printEmployees (Iterator<Employee*>& i) {
             for (i.first(); !i.isDone(); i.next()) {
                 i.currentItem()->print();
             }

List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

printEmployees(forward);
printEmployees(backward);
```