Abstract Factory

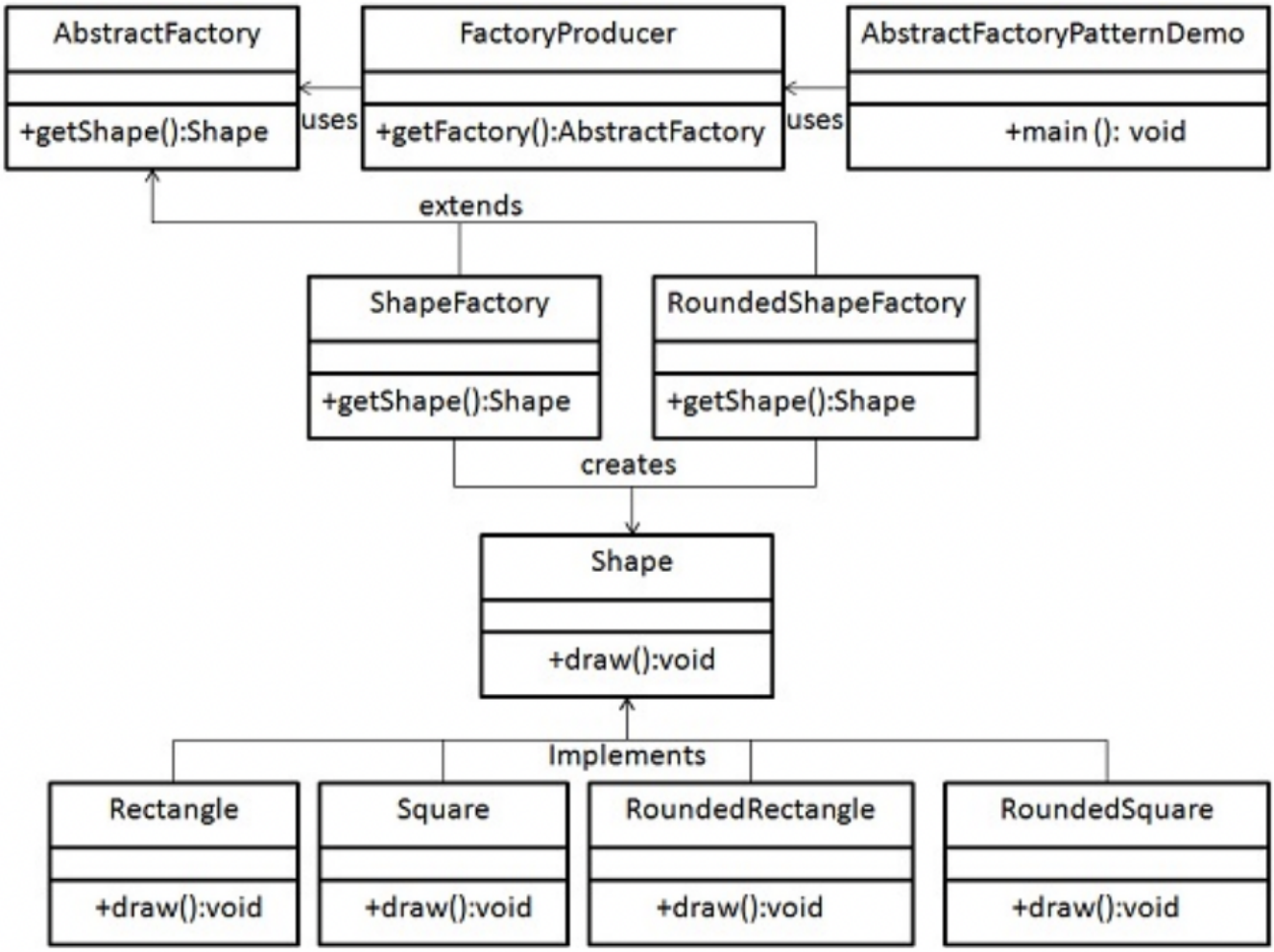
Da utilizzare nel Model

↓ INTENT ↓

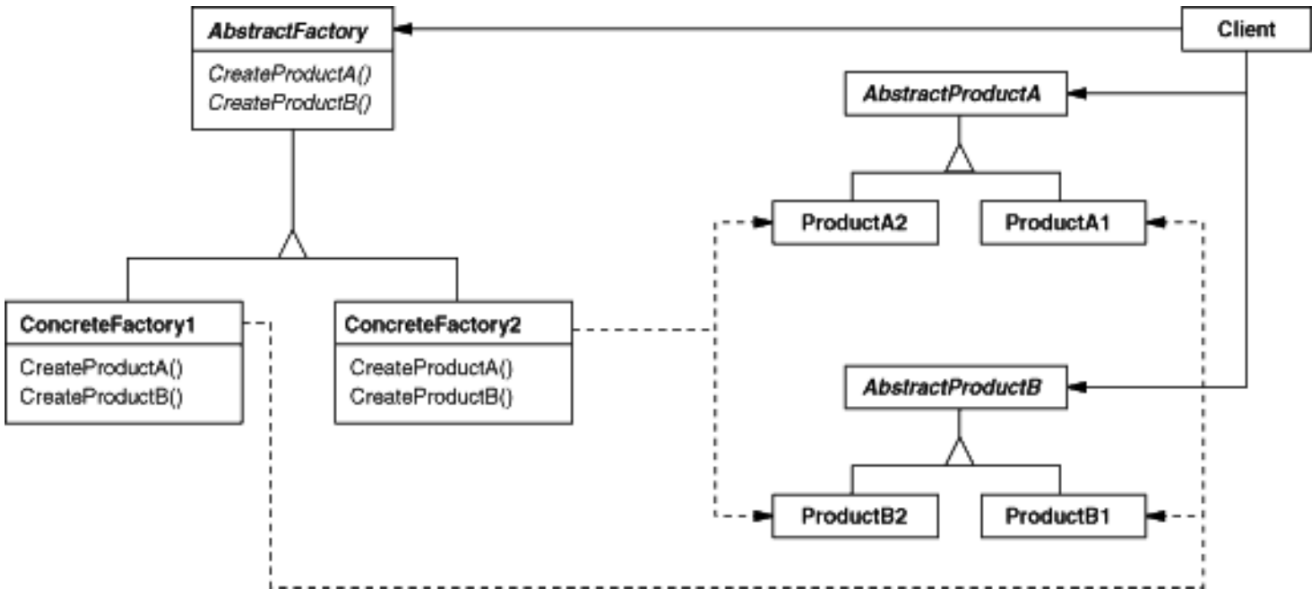
```
Fornisce un'interfaccia per creare famiglie di oggetti correlati o
dipendenti senza specificarne le classi concrete, infatti il costruttore
viene chiamato all'interno della factory.

(nel progetto non usare costruttori ma factory method o abstract factory)
```

↓ MOTIVATION ↓



↓ STRUCTURE ↓



↓ IMPLEMENTATION ↓

---

1. Factory come Singleton.

Un'applicazione in genere richiede solo un'istanza di ConcreteFactory.

Quindi di solito è meglio implementarlo come Singleton.

---

2. Creazione dei Product.

AbstractFactory dichiara solo un'interfaccia per la creazione di prodotti. Spetta alle sottoclassi ConcreteProduct crearle effettivamente definendo un Factory Method per ciascun prodotto.

Questa implementazione, quindi, richiede una nuova sottoclasse concreta per ogni famiglia di prodotti, anche se differiscono leggermente (se sono possibili molte famiglie di prodotti utilizzare il modello Prototype).

↓ EXAMPLE ↓

FactoryProducer ↓

```
public class FactoryProducer {
    public static AbstractFactory getFactory(boolean rounded) {
        if (rounded) {
            return new RoundedShapeFactor();
        } else {
            return new ShapeFactory();
        }
    }
}
```

AbstractFactory ↓

```
public abstract class AbstractFactory {
    abstract Shape getShape(String shapeType);
}
```

ShapeFactory ↓

```
public class ShapeFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }
        return null;
    }

}
```

RoundedShapeFactor ↓

```
public class RoundedShapeFactor extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("RECTANGLE")){
            return new RoundedRectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new RoundedSquare();
        }
        return null;
    }

}
```

Shape ↓

```
public interface Shape {
    void draw();
}
```

Rectangle ↓

```
public class Rectangle implements Shape{

    @Override
    public void draw() {
        System.out.println("RETTANGOLO");
    }

}
```

Square ↓

```
public class Square implements Shape{

    @Override
    public void draw() {
        System.out.println("QUADRATO");
    }

}
```

RoundedRectangle ↓

```
public class RoundedRectangle implements Shape{

    @Override
    public void draw() {
        System.out.println("RETTANGOLO ARROTONDATO");
    }

}
```

RoundedSquare ↓

```
public class RoundedSquare implements Shape{

    @Override
    public void draw() {
        System.out.println("QUADRATO ARROTONDATO");
    }

}
```

Main ↓

```
public static void main(String[] args) {
    AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

    Shape s1 = shapeFactory.getShape("rectangle");
    s1.draw();

    Shape s2 = shapeFactory.getShape("square");
    s2.draw();


    shapeFactory = FactoryProducer.getFactory(true);

    Shape s3 = shapeFactory.getShape("rectangle");
    s3.draw();

    Shape s4 = shapeFactory.getShape("square");
    s4.draw();
}
```