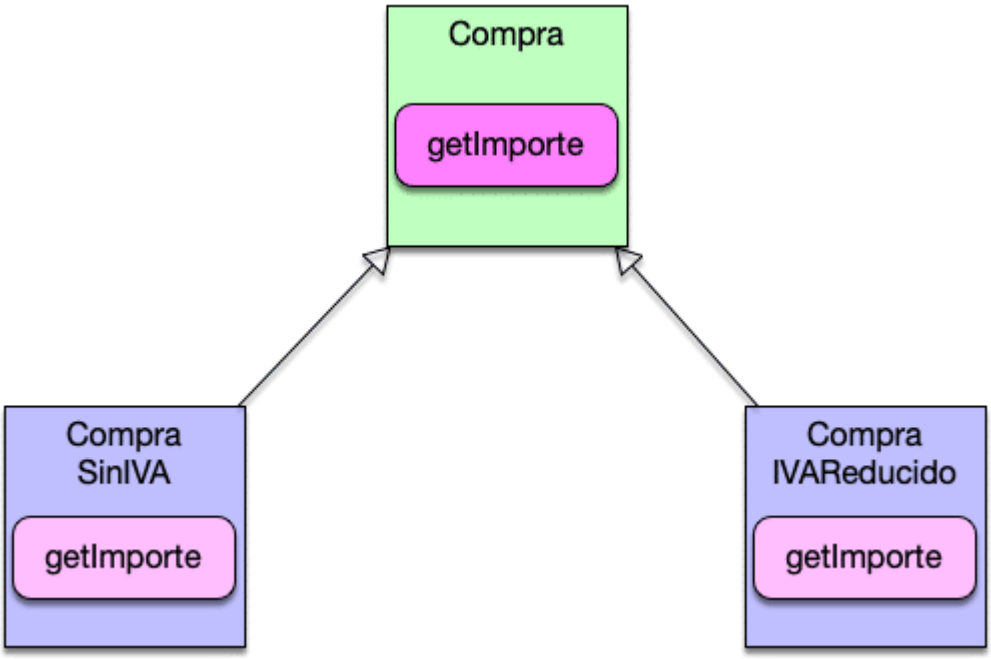
El concepto de Java Strategy Pattern es uno de los conceptos más clásicos ya que se trata de una patrón muy muy utilizado. Para entender el patrón de Strategy tenemos que construirnos en primer lugar una jerarquía de clases que tengan diferente comportamiento entro ellas. Vamos a verlo con la clase Compra a partir de la siguiente jerarquía de clases.



Java Strategy Patern y Herencia

Vamos a construir en primer lugar las clases que necesitamos para cumplimentar este diagrama.

```
package com.arquitecturajava;

public class Compra {

	private String id;
	private String concepto;
	private double importe;
	public String getId() {
		return id;
```

```
	}
	public void setId(String id) {
		this.id = id;
	}
	public String getConcepto() {
		return concepto;
	}
	public void setConcepto(String concepto) {
		this.concepto = concepto;
	}
	public double getImporte() {
		return importe;
	}
	public void setImporte(double importe) {
		this.importe = importe;
	}
	public Compra(String id, String concepto, double importe) {
		super();
		this.id = id;
		this.concepto = concepto;
		this.importe = importe;
	}
	public double getImporteConIVA() {
		return this.getImporte()*1.21;
	}
}
```

```
package com.arquitecturajava;

public class CompraIVAReducido extends Compra {

	public CompraIVAReducido(String id, String concepto, double
```

```
importe) {
		super(id, concepto, importe);
		// TODO Auto-generated constructor stub
	}

	@Override
	public double getImporteConIVA() {
		// TODO Auto-generated method stub
		return Math.round(super.getImporteConIVA()*1.10);
	}

}
```

```
package com.arquitecturajava;

public class CompraSinIVA extends Compra {

	public CompraSinIVA(String id, String concepto, double
importe) {
		super(id, concepto, importe);
		// TODO Auto-generated constructor stub
	}

	@Override
	public double getImporteConIVA() {
		return this.getImporte();
	}

}
```

Una vez tenemos construidas las clases es momento de construir un programa Main o Principal que haga uso de ellas.

```
package com.arquitecturajava;

public class Principal {

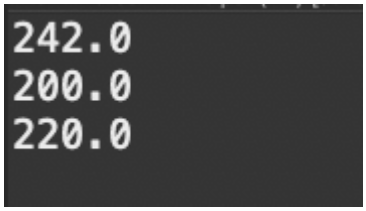
	public static void main(String[] args) {
		Compra c= new Compra("1A","ordenador",200);
		System.out.println(c.getImporteConIVA());

		Compra c2= new CompraSinIVA("2A","ordenador",200);
		System.out.println(c2.getImporteConIVA());
		Compra c3= new
CompraIVAReducido("3A","ordenador",200);
		System.out.println(c3.getImporteConIVA());

	}

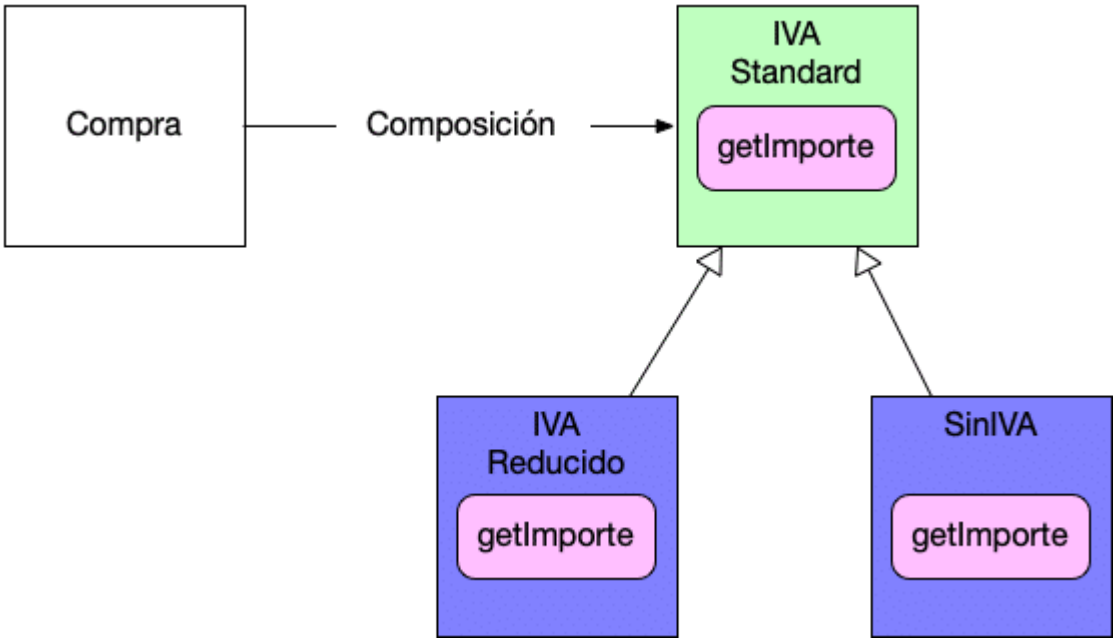
}
```

Si ejecutamos el programa veremos aparecer por la consola los diferentes importes dependiendo del tipo de Compra que se realice.



Java Herencia vs Composición

En muchas ocasiones se dice que la composición es más flexible que la herencia. ¿Es así? .Vamos a verlo a través del uso de Java Strategy Pattern. Este patrón de diseño nos permite realizar la misma operación pero centrándonos en definir la misma funcionalidad del IVA pero a través de delegación.



De esta forma tenemos una única clase Compra y son otras pequeñas clases las que se encargan de dilucidar el tipo de IVA que se aplica.

```
package com.arquitecturajava.ejemplo2;

public class Compra {

	private String id;
	private String concepto;
	private double importe;
	private IVA tipo;
	public String getId() {
		return id;
	}
	public void setId(String id) {
		this.id = id;
	}
	public String getConcepto() {
		return concepto;
```

```
	}
	public void setConcepto(String concepto) {
		this.concepto = concepto;
	}
	public double getImporte() {
		return importe;
	}
	public void setImporte(double importe) {
		this.importe = importe;
	}
	public Compra(String id, String concepto, double importe,IVA
tipo) {
		super();
		this.id = id;
		this.concepto = concepto;
		this.importe = importe;
		this.tipo=tipo;
	}
	public double getImporteConIVA() {
		return tipo.getImporteConIVA(this.getImporte());
	}
}
```

```
package com.arquitecturajava.ejemplo2;

public class IVA {

	public double getImporteConIVA(double importe) {
		return importe *1.21;
	}
}
```

```
package com.arquitecturajava.ejemplo2;

public class IVAReducido  extends IVAStandard{

        public double getImporteConIVA(double importe) {
                return importe;
        }
}
```

```
package com.arquitecturajava.ejemplo2;

public class SinIVA  extends IVAStandard{

        public double getImporteConIVA(double importe) {
                return importe;
        }
}
```

De esta manera conseguimos un funcionamiento idéntico pero realizando una operación de delegación:

```
package com.arquitecturajava.ejemplo2;

public class Principal {

        public static void main(String[] args) {
                Compra c= new Compra("1A","ordenador",200, new
IVAStandard());
                System.out.println(c.getImporteConIVA());
                Compra c2= new Compra("1A","ordenador",200, new
IVAReducido());
                System.out.println(c2.getImporteConIVA());
```

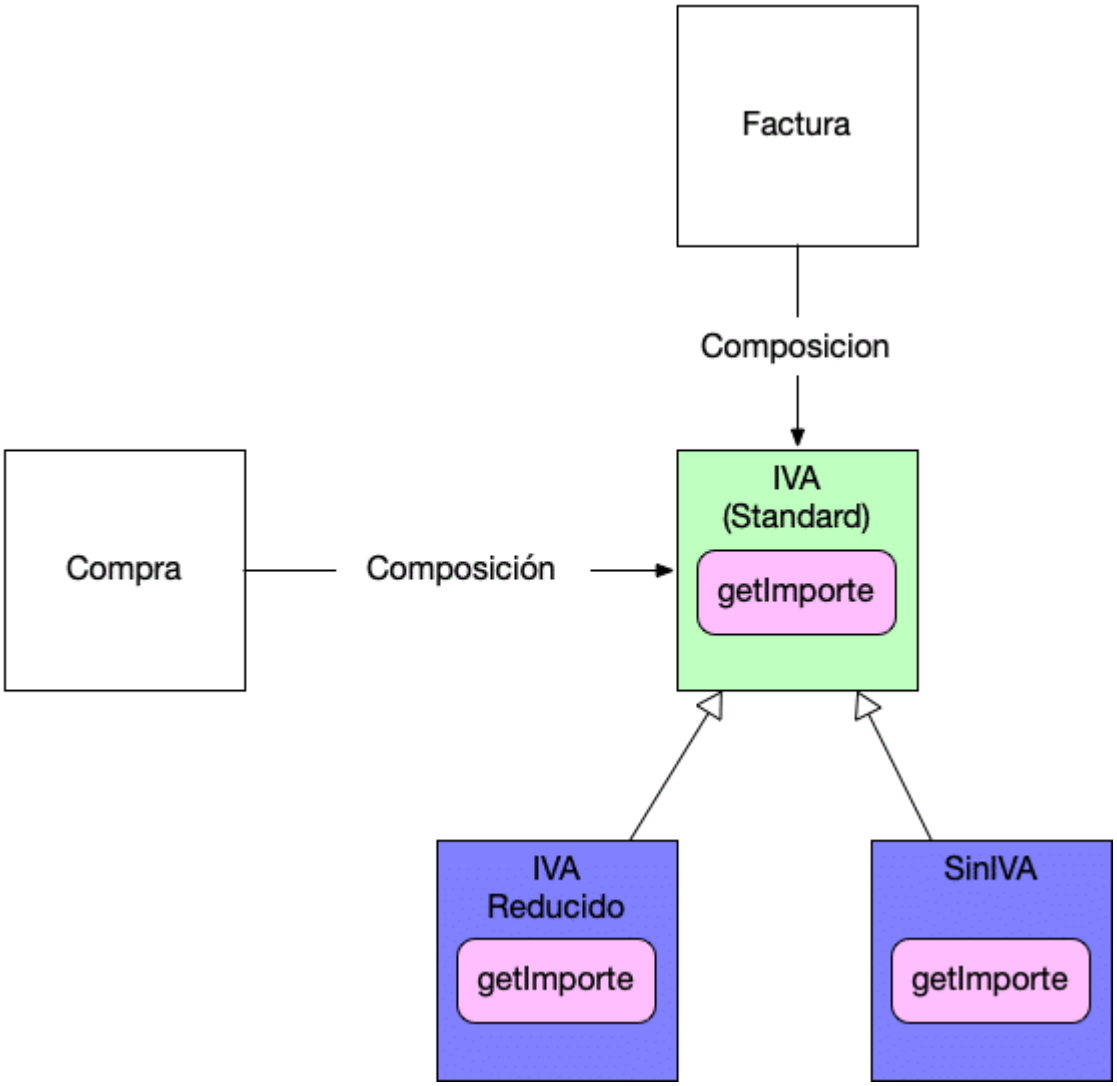
```
                Compra c3= new Compra("1A","ordenador",200, new
IVAReducido());
                System.out.println(c2.getImporteConIVA());


        }

}
```

```
242.0
200.0
220.0
```

El resultado en la consola es idéntico y el código nos pueda parecer redundante. El patrón strategy brilla cuando nosotros tenemos otras clases que puedan necesitar a futuro el mismo tipo de estrategia a la hora de gestionar los importes. Por ejemplo en nuestro caso es muy probable que dispongamos de la clase Factura y necesite la misma gestión del IVA



Vamos a verlo en código:

```
package com.arquitecturajava.ejemplo2;

public class Factura {

	private int numero;
	private String concepto;
	private double importe;
	private String cliente;
```

```
private IVA tipo;

public int getNumero() {
        return numero;
}

public void setNumero(int numero) {
        this.numero = numero;
}

public String getConcepto() {
        return concepto;
}

public void setConcepto(String concepto) {
        this.concepto = concepto;
}

public double getImporte() {
        return importe;
}

public void setImporte(double importe) {
        this.importe = importe;
}

public String getCliente() {
        return cliente;
}

public void setCliente(String cliente) {
```

```
                this.cliente = cliente;
        }

        public Factura(int numero, String concepto, double importe,
String cliente, IVA tipo) {
                super();
                this.numero = numero;
                this.concepto = concepto;
                this.importe = importe;
                this.cliente = cliente;
                this.tipo=tipo;
        }

        public double getImporteConIVA() {

                return tipo.getImporteConIVA(this.getImporte());
        }

}
```

```
package com.arquitecturajava.ejemplo2;

public class Principal2 {

        public static void main(String[] args) {
                Factura f= new Factura(1,"ordenador",200,"pedro", new
IVA());
                System.out.println(f.getImporteConIVA());
                Factura f2= new Factura(1,"ordenador",200,"pedro", new
IVAReducido());
                System.out.println(f2.getImporteConIVA());
                Factura f3= new Factura(1,"ordenador",200,"pedro", new
```

```
SinIVA());
                System.out.println(f3.getImporteConIVA());
        }

}
```

De esta forma reutilizamos la funcionalidad de calculos con IVA usando el Java Strategy Pattern y definiendo el patrón de diseño que nos permite delegar en una jerarquía de algoritmos.

- Java Composicion y la reutilización del código
- Command Pattern en Java y la gestion de tareas
- Java 8 Factory Pattern y su implementación
- Java Patterns