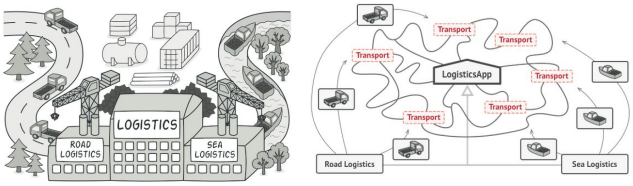
Padrões de Criação

Exemplos Práticos

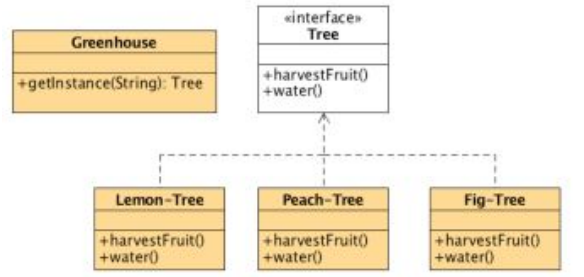
Slides + https://refactoring.guru/design-patterns

Carina Neves 90451

Factory Method:



Fornece uma interface para criar objetos numa superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.

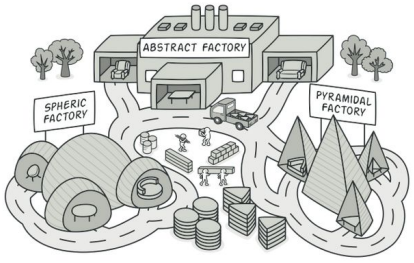



```
interface Arvore {
        void regar();
        void colherFruta();
}
```

```
class Figueira implements Arvore {
        protected Figueira() {System.out.println("Figueira plantada."); }
        public void regar() { System.out.println("Figueira: Regar muito pouco");}
        public void colherFruta() { System.out.println("Hum.. figos!"); }
}

class Pessegueiro implements Arvore {
        protected Pessegueiro() {System.out.println("Pessegueiro plantado."); }
        public void regar() { System.out.println("Pessegueiro: Regar normal"); }
        public void colherFruta() { System.out.println("Boa.. pessegos!"); }
}

class Limoeiro implements Arvore {
        protected Limoeiro() {System.out.println("Limoeiro plantada."); }
        public void regar() { System.out.println("Limoeiro: Regar pouco"); }
        public void colherFruta() { System.out.println("Ahh.. Caipirinha!"); }
}

class Viveiro {
        public static Arvore factory(String pedido){
                if (pedido.equalsIgnoreCase("Figueira")) {
                        return new Figueira(); }
                if (pedido.equalsIgnoreCase("Pessegueiro")) {
                        return new Pessegueiro(); }
                if (pedido.equalsIgnoreCase("Limoeiro"))    {
                         return new Limoeiro(); }
                Else
                        throw new IllegalArgumentException(pedido +" não existente!"); } }
```

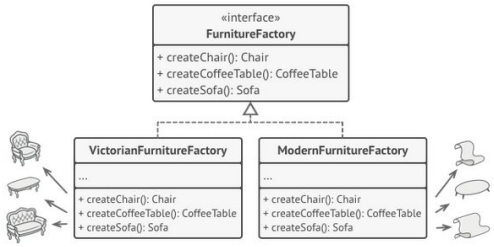
```
public static void main(String[] args) {
        Arvore pomar[] = {
                Viveiro.factory("Figueira"),
                Viveiro.factory("Pessegueiro"),
                Viveiro.factory("Limoeiro")
        };
        for (Arvore a: pomar)
                a.regar();
        for (Arvore a: pomar)
                a.colherFruta();
}
```

Abstact Factory:



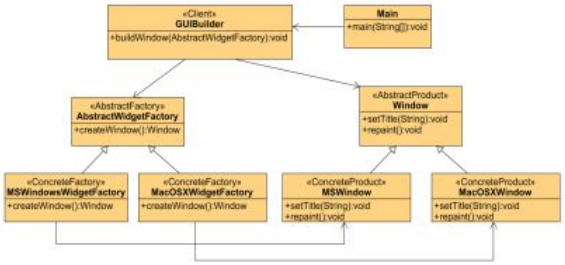
Permite produzir famílias de objetos relacionados sem especificar as suas classes concretas.
Uma fábrica abstrata é uma fábrica que retorna fábricas.

Product families and their variants.



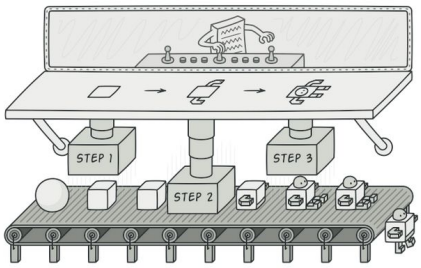
Each concrete factory corresponds to a specific product variant.

```
public class MainTest {
        public static void main(String[] args) {
                GUIBuilder builder = new GUIBuilder();
                If (Platform.currentPlatform()=="MACOSX")
                        builder.buildWindow(new
                MacOSXWidgetFactory());
                else if (Platform.currentPlatform()=="WIN")
                        builder.buildWindow(new
                MsWindowsWidgetFactory());
                else //...
        }
}
```

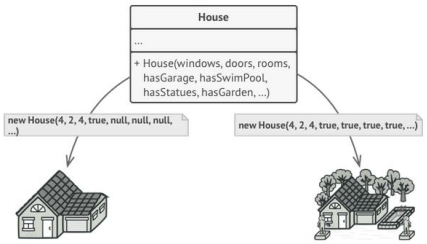


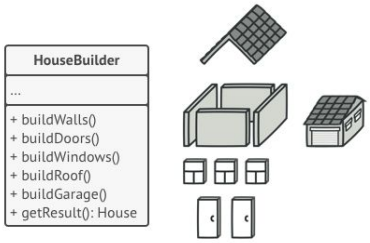
```
public class GUIBuilder {
	public void buildWindow(AbstractWidgetFactory widgetFactory) {
		Window window = widgetFactory.createWindow();
		window.setTitle("New Window");
	}
}
```

Builder:



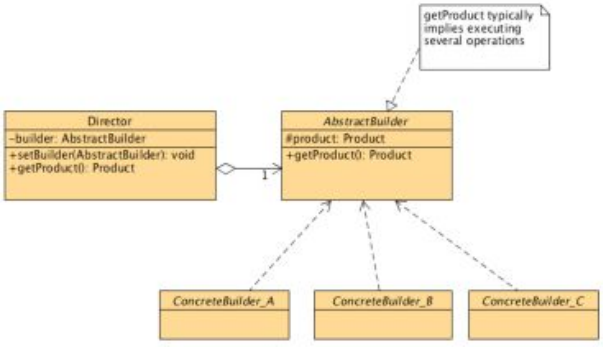
Permite construir objetos complexos passo a passo, permite produzir diferentes tipos e representações de um objeto usando o mesmo código de construção.





1:

```
class Pizza {  /* "Product" */
	private String dough;
	private String sauce;
	private String topping;
	public void setDough(String dough)
		{ this.dough = dough; }
	public void setSauce(String sauce)
		{ this.sauce = sauce; }
	public void setTopping(String topping) {
this.topping = topping; }
}

abstract class PizzaBuilder {
	protected Pizza pizza;  /* "Abstract Builder" */
	public Pizza getPizza() { return pizza; }
```



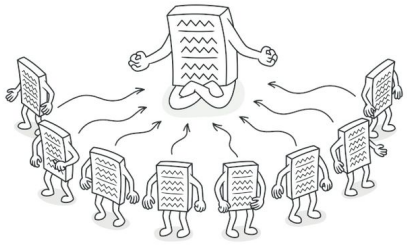
```
	public void createNewPizzaProduct() { pizza = new Pizza(); }
	public abstract void buildDough();
	public abstract void buildSauce();
	public abstract void buildTopping();
}

/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
	public void buildDough() { pizza.setDough("cross"); }
	public void buildSauce() { pizza.setSauce("mild"); }
	public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
	public void buildDough() { pizza.setDough("pan baked"); }
	public void buildSauce() { pizza.setSauce("hot"); }
	public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}

class Waiter { /* "Director" */
	private PizzaBuilder pizzaBuilder;
	public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
	public Pizza getPizza() { return pizzaBuilder.getPizza(); }
	public void constructPizza() {
		pizzaBuilder.createNewPizzaProduct();
		pizzaBuilder.buildDough();
		pizzaBuilder.buildSauce();
		pizzaBuilder.buildTopping();
	}
}

/* A customer ordering a pizza. */
class BuilderExample {
	public static void main(String[] args) {
		Waiter waiter = new Waiter();
		PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
		PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();
		waiter.setPizzaBuilder( hawaiian_pizzabuilder );
		waiter.constructPizza();
		Pizza pizza = waiter.getPizza();
		waiter.setPizzaBuilder( spicy_pizzabuilder );
		waiter.constructPizza();
		pizza = waiter.getPizza();
	}
}
```

2:

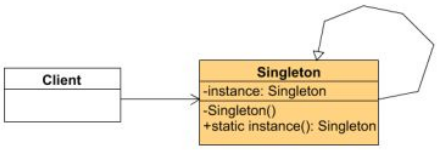
```
public class NutritionFacts { // Builder Pattern
	private final int servingSize;
	private final int servings;
	private final int calories;
	private final int fat;
	private final int sodium;
	private final int carbohydrate;

	public static class Builder {
		// Required parameters
		private final int servingSize;
		private final int servings;
		// Optional parameters - initialized to default values
		private int calories = 0;
		private int fat = 0;
		private int carbohydrate = 0;
		private int sodium = 0;

		public Builder(int servingSize, int servings) {
			this.servingSize = servingSize;
			this.servings = servings;
		 }
		public Builder calories(int val) {
			calories = val;
			return this;
		}
		public Builder fat(int val) {
			fat = val;
			return this;
		}
		public Builder carbohydrate(int val) {
			carbohydrate = val;
			return this;
		}
		public Builder sodium(int val) {
			sodium = val;
			return this;
		}
		public NutritionFacts build() { return new NutritionFacts(this); }

	} // end of class Builder

	private NutritionFacts(Builder builder) {
		servingSize = builder.servingSize;
		servings = builder.servings;
		calories = builder.calories;
		fat = builder.fat;
		sodium = builder.sodium;
```

```
            carbohydrate = builder.carbohydrate;
        }
}
```

Singleton:

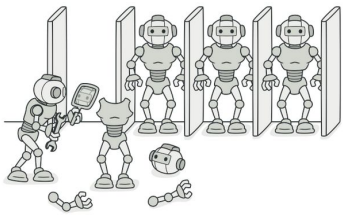


Permite garantir que uma classe tem apenas uma instância, fornecendo um ponto de acesso global a essa instância.



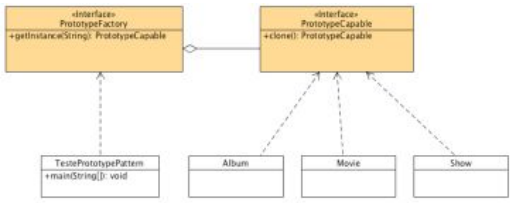

```
class Singleton {
        private String name;
        static private Singleton instance = new
Singleton("Ermita");
        private Singleton(String name) {
            this.name = name;
        }
        static public Singleton getInstance() {
            return instance;
        }
        @Override
        public String toString() {
            return name;
        }
}
```

Prototype:



Pre-built prototypes can be an alternative to subclassing.

Permite copiar objetos existentes sem tornar seu código dependente de suas classes.




```
public interface PrototypeCapable extends Cloneable {
	public PrototypeCapable clone() throws
CloneNotSupportedException;
}

public class Album implements PrototypeCapable {
	private String name = null;
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	@Override
	public Album clone() throws CloneNotSupportedException {
		System.out.println("Cloning Album object..");
		return (Album) super.clone();
	}
	@Override
	public String toString() {
		return "Album";
	}
}

public class PrototypeFactory {
	public static enum ModelType {
		MOVIE, ALBUM, SHOW;
	}

	private static Map<ModelType, PrototypeCapable> prototypes = new HashMap<>();

	static {
		prototypes.put(ModelType.MOVIE, new Movie());
		prototypes.put(ModelType.ALBUM, new Album());
		prototypes.put(ModelType.SHOW, new Show());
	}
	public static PrototypeCapable getInstance(ModelType s)
		throws CloneNotSupportedException {
			return ((PrototypeCapable) prototypes.get(s)).clone();
		}
}

public class TestPrototypePattern {
	public static void main(String[] args) {
		try {
			PrototypeCapable proto;
			proto = PrototypeFactory.getInstance(ModelType.MOVIE);
			System.out.println(proto);
```

```
                proto = PrototypeFactory.getInstance(ModelType.ALBUM);
                System.out.println(albumPrototype);
                proto = PrototypeFactory.getInstance(ModelType.SHOW);
                System.out.println(proto);
            }
            catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
        }
}
```

Resumo das funcionalidades:

Factory method- Cria uma instância de várias classes derivadas
Abstract factory- Cria uma instância de várias famílias de classes
Builder- Separa a construção do objeto de sua representação
Singleton- Uma classe da qual apenas uma única instância pode existir
Prototype- Uma instância totalmente inicializada para ser copiada ou clonada