Esempi di alcune design pattern

Factory

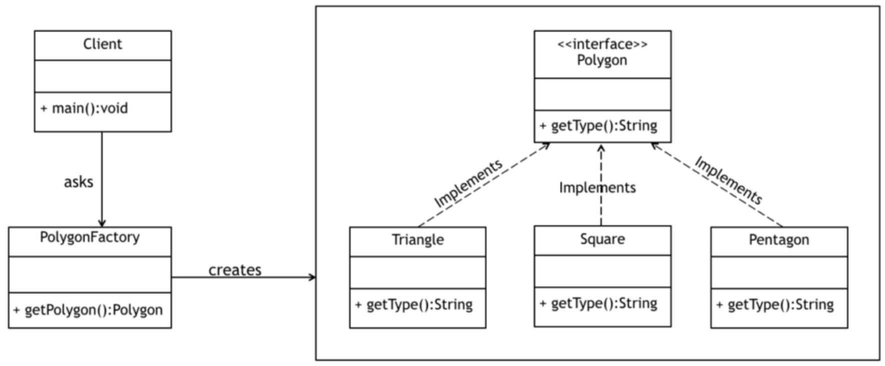

Figure 1: Factory

Come fare per nascondere i dettagli di come costruire gli oggetti?

```
package it.uniud.poo.designPatterns.factory;

public class ClientFactory {
       public static void main( String a[] ) {

           /**
            * con la factory si nascondono i dettagli relativi alla costruzione
            * degli oggetti e soprattutto quelli relativi al tipo reale utilizzato
            */
           PolygonFactory theFactory = new PolygonFactory();
           Polygon p = theFactory.createPolygon( 3 );
           System.out.format( "Polygon: %s", p.getType() );
       }
}
```

```
package it.uniud.poo.designPatterns.factory;

public class PolygonFactory {

    public Polygon createPolygon( int numberOfSides) {
```

```java
        if(numberOfSides == 3) {
            return new Triangle();
        }
        if(numberOfSides == 4) {
            return new Square();
        }
        if(numberOfSides == 5) {
            return new Pentagon();
        }

        throw new RuntimeException( "unhandled polygon" );
    }
}
```

```java
package it.uniud.poo.designPatterns.factory;

// tratto da http://www.baeldung.com/creational-design-patterns

public interface Polygon {
    String getType();
}
```

```java
package it.uniud.poo.designPatterns.factory;

public class Triangle implements Polygon {
    @Override
    public String getType() {
        return "I am a trangle";
    }
}
```

```java
package it.uniud.poo.designPatterns.factory;

public class Square implements Polygon {
    @Override
    public String getType() {
        return "I am a square";
    }
}
```

```
package it.uniud.poo.designPatterns.factory;

public class Pentagon implements Polygon {
    @Override
    public String getType() {
        return "I am a pentagon";
    }
}
```

Abstract Factory

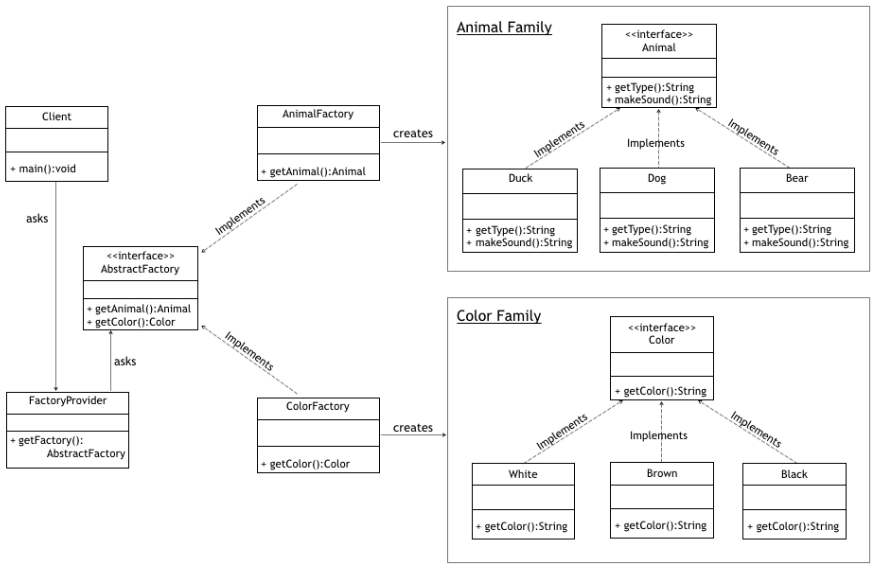

Figure 2: Abstract Factory

Come fare per nascondere i dettagli di come costruire varie tipologie di oggetti? cioè disporre di varie factory?

```
package it.uniud.poo.designPatterns.abstractFactory;

public class ClientAbstractFactory {

    public static void main( String[] args ) {
```

```java
        //creating a brown dog
        FactoryProvider fp = new FactoryProvider();
        AbstractFactory theAnimalFactory = fp.getFactory( "Animal" );
        AbstractFactory theColorFactory  = fp.getFactory( "Color" );

        Animal toy = theAnimalFactory.createAnimal( "Dog" );
        Color color = theColorFactory.createColor( "Brown" );

        System.out.format( "\n%s (sound %s);  %s",
        toy.getAnimalName(), toy.makeSound(), color.getColor() );

        toy = theAnimalFactory.createAnimal( "cat" );
        color = theColorFactory.createColor( "white" );
        System.out.format( "\n%s (sound %s);  %s",
        toy.getAnimalName(),
        toy.makeSound(), color.getColor() );

        /**
         * il client deve solo sapere cosa dire per chiedere di ottenere
         * una certa factory
         * (es. fp.getFactory("Animal"))
         * e cosa dire per chiedere un certo animale o un certo colore.
         * Il client non sa quali tipi reali vengono usati per gli animali
         * e per i colori.
         * Non sa neppure quali sono le reali factory che vengono usate.
         * Che quindi possono venir modificati senza dover richiedere
         * modifiche al client.
         *
         * Abstract Factory e' comoda quando abbiamo varie gerarchie
         * di tipo (animali e colori)
         * che dobbiamo manipolare.
         */

    }
}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;

import java.util.Objects;
```

```java
public class FactoryProvider {
    public AbstractFactory getFactory( String choice ) {
        Objects.requireNonNull(choice );
        if ( choice.equalsIgnoreCase( "animal" )) {
            return new AnimalFactory();
        } else if (choice.equalsIgnoreCase( "color" )){
            return new ColorFactory();
        } else {
            throw new RuntimeException();
        }
    }
}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;

public interface AbstractFactory {
    Animal createAnimal (String animalType);
    Color createColor (String colorType);

}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;

import java.util.Objects;

public class AnimalFactory implements AbstractFactory {

    @Override
    public Animal createAnimal(String animalType) {
        Objects.requireNonNull( animalType );
        if (animalType.equalsIgnoreCase("dog")) {
            return new Dog();
        } else if (animalType.equalsIgnoreCase("duck")) {
            return new Duck();
        } else if (animalType.equalsIgnoreCase( "cat" )){
            return new Cat();
        }
        throw new RuntimeException( "animal does not exist: "+animalType );
    }

    @Override
    public Color createColor(String color){
        throw new UnsupportedOperationException();
    }

}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;

import java.util.Objects;

public class ColorFactory implements AbstractFactory {

    @Override
    public Color createColor( String colorType ) {
        Objects.requireNonNull( colorType );
        if (colorType.equalsIgnoreCase( "brown" )) {
            return new Brown();
        } else if (colorType.equalsIgnoreCase( "white" )) {
            return new White();
        }
        throw new RuntimeException( "Color does not exist: " + colorType );
    }

    @Override
    public Animal createAnimal( String toyType ) {
        throw new UnsupportedOperationException();
    }

}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;

// tratto da http://www.baeldung.com/creational-design-patterns

public interface Animal {
    String getAnimalName();
    String makeSound();
}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;

public class Cat implements Animal {
    @Override
    public String getAnimalName() {
        return "I'm a cat";
    }

    @Override
    public String makeSound() {
        return "miau";
    }
}
package it.uniud.poo.designPatterns.abstractFactory;

public class Dog implements Animal {
    @Override
    public String getAnimalName() {
        return "I'm a dog";
    }

    @Override
    public String makeSound() {
        return "bau";
    }
}

package it.uniud.poo.designPatterns.abstractFactory;

public class Duck implements Animal {
    @Override
    public String getAnimalName() {
        return "I'm a duck";
    }

    @Override
    public String makeSound() {
        return "squeck";
    }
}
```

```java
package it.uniud.poo.designPatterns.abstractFactory;
```

```java
public interface Color {
    String getColor();
}
package it.uniud.poo.designPatterns.abstractFactory;

public class Brown implements Color {
    @Override
    public String getColor() {
        return "this is brown";
    }
}

package it.uniud.poo.designPatterns.abstractFactory;

public class White implements Color {
    @Override
    public String getColor() {
        return "this is white";
    }
}
```

Builder

Come fare per avere flessibilità nel modo con cui si possono costruire gli oggetti?

```java
package it.uniud.poo.designPatterns.builder;

public class ClientBuilder {

    public static void main(String[] args) {
        BankAccount newAccount = new BankAccount
                .BankAccountBuilder("John", "22738022275")
                .withEmail("john@example.com")
                .wantNewsletter(true)
                .build();

        System.out.println("Name: " + newAccount.getName());
        System.out.println("AccountNumber:" + newAccount.getAccountNumber());
        System.out.println("Email: " + newAccount.getEmail());
        System.out.println("Want News letter?: " + newAccount.isNewsletter());
        /**
         * notare l'uso dello stile fluent per costruire incrementalmente l'istanza
         * e la distinzione tra campi obbligatori (name e accountNumber)
         * e campi opzionali (email e newsletter).
         *
         * Ovviamente il metodo chiave e' build(), altrimenti le informazioni non
         * vengono trasferite dal builder all'account.
         */
    }
}
```

```
package it.uniud.poo.designPatterns.builder;

// tratto da http://www.baeldung.com/creational-design-patterns

public class BankAccount {

    private String name;
    private String accountNumber;
    private String email;
    private boolean newsletter;


    private BankAccount(BankAccountBuilder builder) {
        this.name = builder.name;
        this.accountNumber = builder.accountNumber;
        this.email = builder.email;
        this.newsletter = builder.newsletter;
    }


    public String getAccountNumber() {
        return accountNumber;
    }

    public void setAccountNumber( String accountNumber ) {
        this.accountNumber = accountNumber;
    }

    public String getName() {
        return name;
    }

    public void setName( String name ) {
        this.name = name;
```

```
}

public String getEmail() {
    return email;
}

public void setEmail( String email ) {
    this.email = email;
}

public boolean isNewsletter() {
    return newsletter;
}

public void setNewsletter( boolean newsletter ) {
    this.newsletter = newsletter;
}

/**
 * inner class for the builder
 */
public static class BankAccountBuilder {
    public String name;
    public String accountNumber;
    public boolean newsletter;
    public String email;

    public BankAccountBuilder(String name, String accountNumber) {
        this.name = name;
        this.accountNumber = accountNumber;
    }

    public BankAccountBuilder withEmail(String email) {
        this.email = email;
        return this;
    }

    public BankAccountBuilder wantNewsletter(boolean newsletter) {
        this.newsletter = newsletter;
        return this;
    }

    //the actual build method that prepares and returns a BankAccount object
    public BankAccount build() {
        return new BankAccount(this);
    }
    }
}
```

Observer

Come fare per slegare chi produce dei dati o li aggiorna da chi li usa?

```
package it.uniud.poo.designPatterns.observer;

public class ClientObserver {

    public static void main( String[] args ) {

        // la news agency produce le notizie
        NewsAgency agency = new NewsAgency();

        // il canale televisivo le osserva, rilancia
        NewsChannel observer = new NewsChannel( "rai1");

        agency.addObserver( observer );

        // altro observer
        agency.addObserver( new NewsChannel( "rai2") );
        agency.setNews( "news di oggi: 30 e lode a tutti" );

        /**
         * notare come l'oggetto Observable (la NewsAgency)
         * non sa cosa fare dal punto di vista di come "usare" le news, che
         * e' una responsabilita' demandata al canale (l'osservatore)
         *
         */
    }
}
```

```
package it.uniud.poo.designPatterns.observer;


import java.util.ArrayList;
import java.util.List;
```

```java
public interface Observer {

    public void update( Object data ) ;
}
```

```java
package it.uniud.poo.designPatterns.observer;

public interface Observable {
    public void addObserver( Observer obs ) ;
}
```

```java
package it.uniud.poo.designPatterns.observer;

import java.util.ArrayList;
import java.util.List;

public class NewsAgency implements Observable {

    private String news;

    private List<Observer> observers = new ArrayList<>();

    @Override
    public void addObserver( Observer obs ) {
        this.observers.add( obs );
    }

    public void setNews( String news ) {
        this.news = news;
        updateObservers();
    }

    private void updateObservers() {
        for (Observer obs : this.observers) {
            obs.update( this.news );
        }
    }

}
```

```
package it.uniud.poo.designPatterns.observer;


public class NewsChannel implements Observer {

    String name ="";
    NewsChannel (String name){
        this.name=name;
    }
    @Override
    public void update( Object data ) {
        System.out.format("\nSono %s: ricevuto la news %s", this.name, data );
    }
}
```

Decorator

Come fare per avere una macchina sia sportiva che di lusso?

```
package it.uniud.poo.designPatterns.decorator;

public class ClientDecorator {

    public static void main(String[] args) {

        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println("\n*****");

        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
        sportsLuxuryCar.assemble();
        /**
         * in pratica si riesce a fare un mix-in di 2+ sotto-classi
         * cosa che non si puo' fare con una tipologa basata solo sulla gerarchia
         *
```

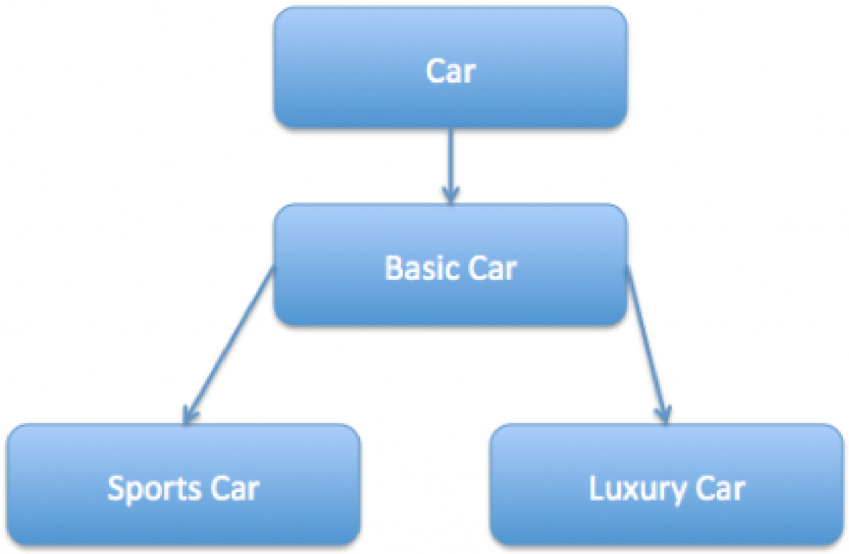

Figure 3: Inheritance

```
         * NB il client puo' decidere quali decoratori usare e anche
         * l'ordine con cui applicarli.
         *
         */

        System.out.println("\n*****");
        Car luxurySportsCar = new LuxuryCar(new SportsCar(new BasicCar()));
        luxurySportsCar.assemble();

    }

}



package it.uniud.poo.designPatterns.decorator;

// tratto da https://www.journaldev.com/1540/decorator-design-pattern-in-java-example

public interface Car {
    public void assemble();
}
```

```java
package it.uniud.poo.designPatterns.decorator;

public class BasicCar  implements  Car {

    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }
}
```

```java
package it.uniud.poo.designPatterns.decorator;

public class CarDecorator implements Car {

    // NB  protected invece di private

    protected Car decoratedCar;

    public CarDecorator( Car c ) {
        this.decoratedCar = c;
    }

    @Override
    public void assemble() {
        this.decoratedCar.assemble();
    }

}
```

```java
package it.uniud.poo.designPatterns.decorator;

public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}

package it.uniud.poo.designPatterns.decorator;

public class SportsCar extends CarDecorator {

    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}
```