CORSO
DESIGN PATTERN

I pattern Creational

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Pattern Creational

- **Scopo dei pattern di creazione**
  - Rendere i componenti di un sistema, che usa determinati oggetti, indipendenti da come tali oggetti siano stati creati, composti o rappresentati
    - Nasconde la conoscenza di quali classi sono realmente istanziate
    - Nasconde i dettagli sulla creazione / composizione
    - Riduce accoppiamento e aumenta la flessibilità

Elenco Pattern

1. Factory method
2. Abstract Factory
3. Prototype
4. Builder
5. Singleton

1) Factory Method

Esempio:

- In un framework, per la manipolazione di elementi cartografici, si vogliono modellare le classi:
  - **Elemento → MapElement**, generico elemento della mappa (es. luogo, città, collegamento..)
  - **Strumento → ElementHandler**, che fornisce le operazioni comuni di manipolazione degli Elementi

Requisiti:

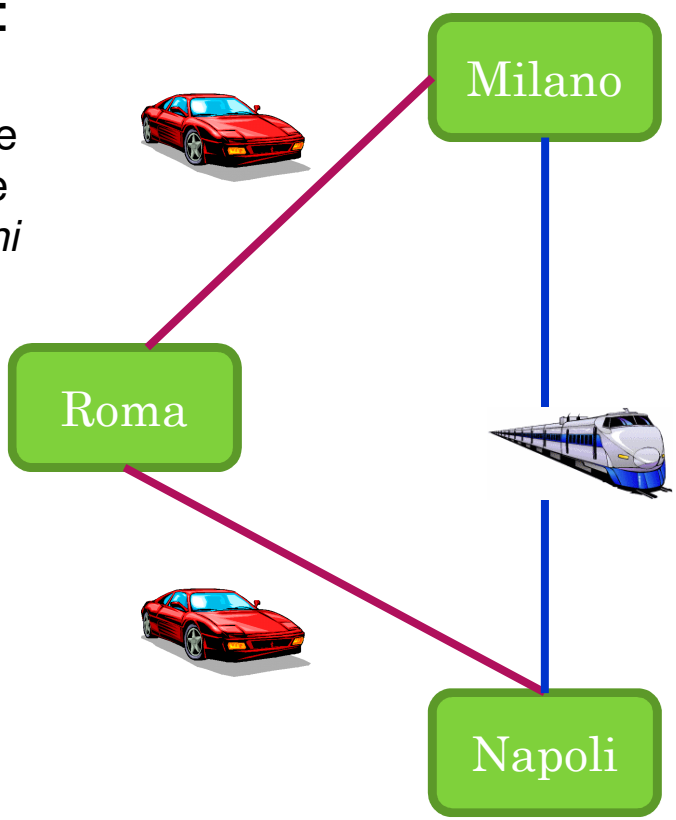
- Lo Strumento deve usare l'Elemento senza conoscere il tipo specifico ma solo l'interfaccia di funzionamento.
- Gli Elementi potrebbero diventare molti in futuro
- Il Framework non vuole legarsi ai diversi tipi di Elementi

Segue esempio d'uso →

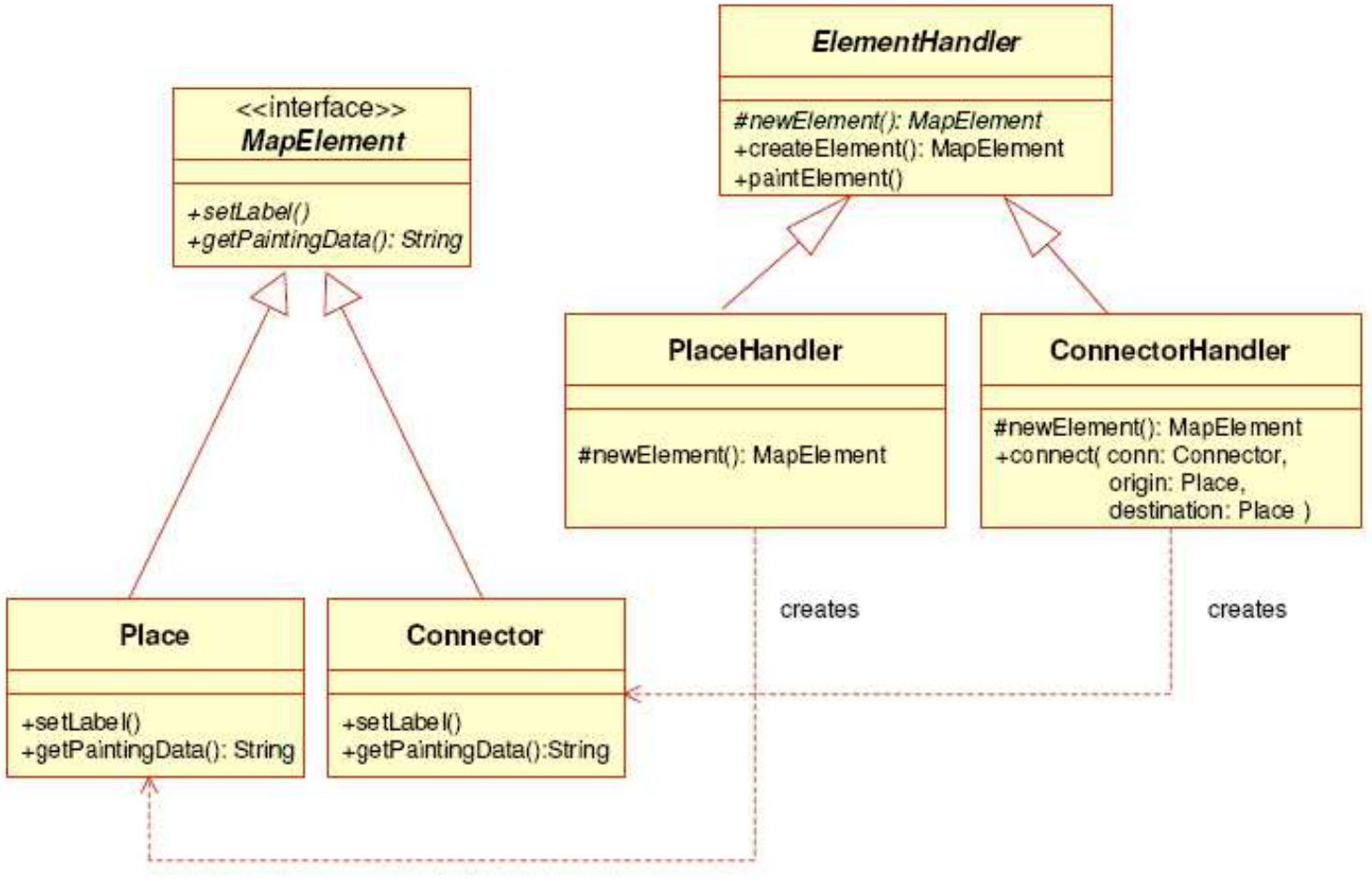
1) Factory Method

Esempio d'uso:

Vogliamo disegnare questi *luoghi* con le relative *connessioni*



Class Diagram esempio



- L'**ElementHandler** espone metodi pubblici di creazione/gestione elementi
- Gli elementi vengono creati dai **ConcreteHandler** e usati come **MapElement**

1) Factory Method

- **Soluzione:**

  Il pattern "*Factory Method*" suggerisce di delegare la creazione di ogni particolare tipo di Element all'Handler degli elementi e non al framework.

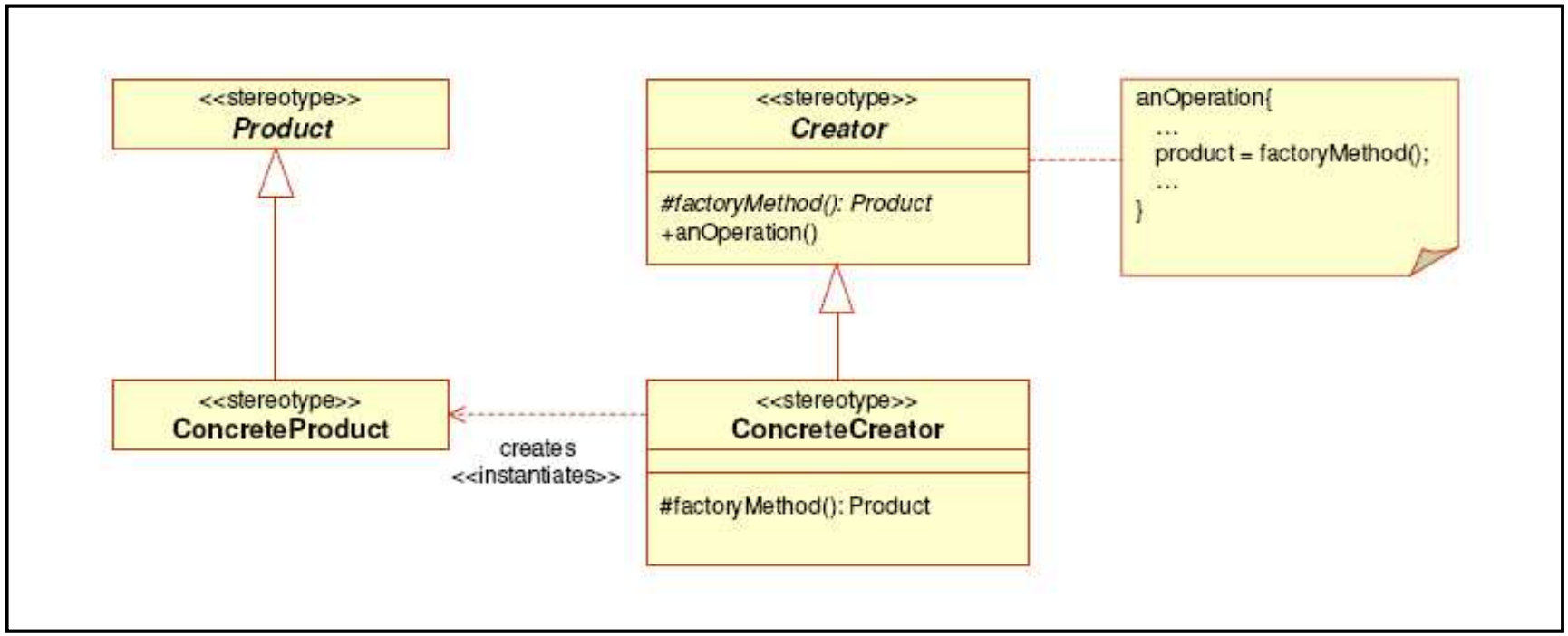
  Il metodo di creazione restituisce oggetti generici di tipo Elemento (MapElement) e viene specializzato dagli Handler concreti per creare le specifiche istanze delle classi che gestiscono.

  - Il framework crea un Handler per un dato MapElement
  - L'handler crea l'elemento e fornisce i metodi per gestirlo.

1) Factory Method

- **Problema:**
  - Si vuole creare un oggetto di cui staticamente si conosce l'interfaccia (o la super classe) mentre la classe effettiva viene decisa anche dinamicamente.
  - Tipica situazione che compare nei framework
- **Soluzione:**
  - Definire una classe che gestisce oggetti
  - Incapsulare la logica di creazione in un metodo astratto per poi specializzarla nelle sottoclassi
  - Separare chi crea oggetti da chi li usa.
  - Si sfrutta essenzialmente il polimorfismo del linguaggio

1) Factory Method Class Diagram



- Il **Creator** espone un metodo pubblico che viene usato dal **Consumer** per ottenere i prodotti desiderati.
- Gli oggetti vengono creati come **ConcreteProduct**, ma usati come **Product**

1) Factory Method

Conseguenze:

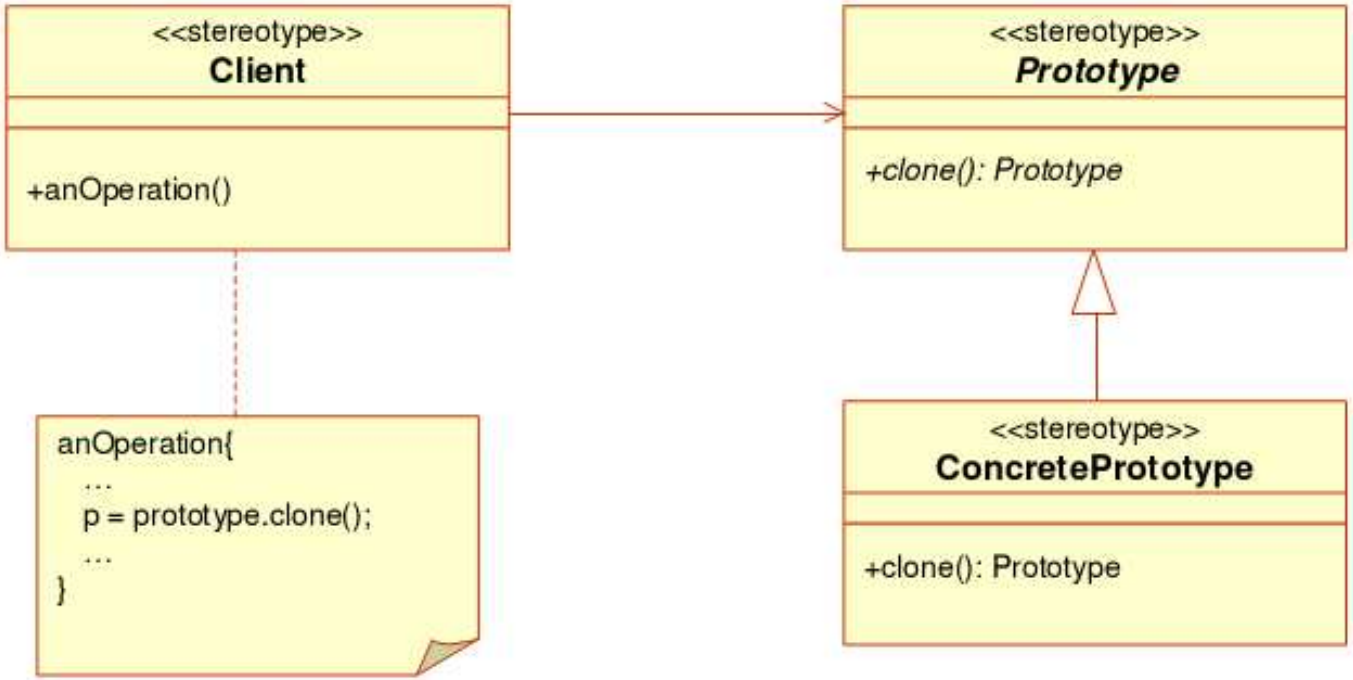
- Grazie al polimorfismo, ogni oggetto concreto opera secondo la sua "natura" senza che venga "tradita"
- La struttura è facilmente estensibile aggiungendo ConcreteProduct e ConcreteCreator
- Per ogni nuovo Product, bisognerà predisporre un nuovo Creator.

2) Prototype

- **Problema:**
  - Disporre di un oggetto di cui si conosce l'interfaccia, ma non la reale classe.
  - Gli oggetti da creare sono molto complessi e numerosi
  - Le classi da istanziare non sono note a priori→ FactoryMethod è scomodo!!
- **Soluzione:**
  - Creare dei prototipi da cui generare successive copie
  - Disporre una mappa di prototipi (eventualmente modificabile)
  - Predisporre una factory che gestisce la mappa

2) Prototype Class Diagram

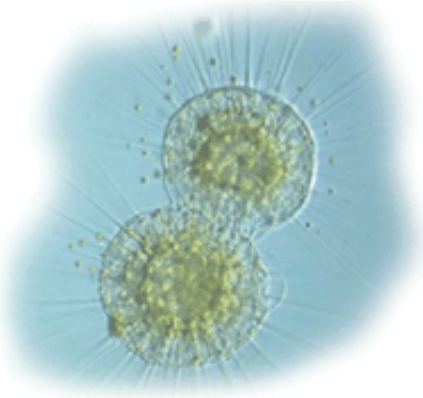
- L'idea è quella di realizzare un prodotto dal suo prototipo.
- Questo è giustificato spesso dalla complessità intrinseca dell'oggetto.

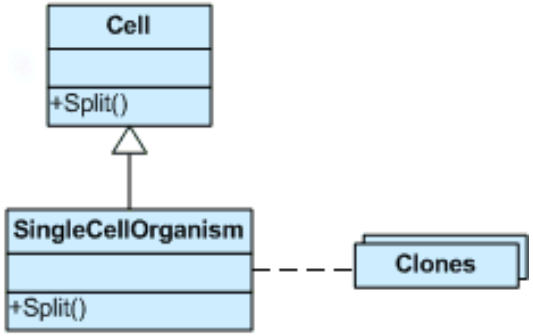


Nel libro del GoF, il pattern viene presentato facendo ricorso alla clonazione, così è applicabile anche a linguaggi senza meta informazioni (C++)

Esempio

- Nel processo di **mitosi**, divisione cellulare, il **nucleo cellulare si divide** in modo tale che ciascuna cellula figlia **abbia esattamente lo stesso patrimonio** cromosomico della cellula madre.





- Durante il processo di divisione mitotica, i cromosomi della cellula madre **vengono duplicati** e distribuiti in due cellule figlie identiche tra di loro

2) Prototype

- **Passi per realizzare il Prototype:**
  - Definire la superclasse (Prototype) dei prototipi.
    - Impostare il metodo di clonazione generico
  - Realizzare i prototipi concreti
    - con i rispettivi criteri di clonazione
  - Creare una classe Creator che si occupa della creazione degli oggetti della categoria Prototype
    - si occuperà della gestione della mappa di prototipi
  - Il Client può richiedere un oggetto a condizione che ne esiste il prototipo
    - oppure la nuova richiesta – se si può evadere – produce l'aggiornamento della mappa dei prototipi

Esempio


```
public abstract class Prototype implements Cloneable{
   protected ComplexStructure struct; // struttura complessa
   public Prototype() {
        this.struct = Utility.buildStruct();
   }
   public Object clone() {
        Prototype pt = null;
        try{
                pt = (Prototype)super.clone();
                pt.struct = this.struct;
        }catch(CloneNotSupportedException e){}
        return pt;
   }
}
```

Esempio


```
public class Creator {
   private HashMap<String, Prototype> protos =
                                   new HashMap<String, Prototype>();
   public void registerPrototype(String key, Prototype proto) {
        protos.put(key, proto);
   }
   public Prototype createObject(String key)
                                          throws TypeException{
        Prototype proto = (Prototype)protos.get(key);
        if (proto == null)
                throws new TypeException("Unsupported Type");
        return proto.clone();
   }
}
```

Esempio


```
public class Client{
   public static void main(String[] args) {
        Creator creator = new Creator();
        // registro il prototipo
        creator.registerPrototype("myProto",new MyPrototype());
        // ottengo le istanze per clonazione
        for(int i=0;i<10;i++){
                System.out.println(creator.getObject("myProto"));
        }
   }
}
```

La classe **`MyPrototype`** è sottoclasse di **`Prototype`** e prevede il suo metodo di clonazione.
Se i tipi dei prototipi fossero configurati in un file properties, nel Client si potrebbe evitare la creazione esplicita dell'oggetto prototipo.

Idioma in java

- In java la tecnica della *Reflection* (creazione di oggetti tramite le meta informazioni del linguaggio) trasforma il Prototype in una sorta di idioma

```
public class Creator {
   public AbstractObject createObject(String clazz) {
        Class cls = Class.forName(clazz);
        return (AbstractObject) cls.newInstance();
   }
}
```

- Il metodo restituisce il generico tipo, a condizione che la classe `clazz` appartenga alla gerarchia di **`AbstractObject`**.

2) Prototype

Conseguenze:

- E' particolarmente utile quando le inizializzazioni risultano molto "costose" , lunghe, complesse e quando i parametri di inizializzazione non "variano molto".
- Non sfrutta l'ereditarietà, ma richiede un'inizializzazione

  → si creano oggetti solo dopo aver creato il prototipo
  - Il FactoryMethod invece sfrutta l'ereditarietà e non richiede inizializzazioni
- Se si dispone della Java Reflection si può realizzare un meccanismo di creazione fortemente dinamico, non basato sulla classica clonazione

3) Abstract Factory

Esempio:

- Si vuole realizzare un negozio di vendita di sistemi Hi-Fi, dove si eseguono dimostrazioni dell'utilizzo dei prodotti.
- Esistono due famiglie di prodotti, basate su tecnologie diverse:
  - supporto di tipo nastro (**tape**)
  - supporto di tipo digitale (**CD**).

- Ogni famiglia è composta da:
  - supporto (tape o cd)
  - masterizzatore (recorder)
  - riproduttore (player).

3) Abstract Factory

- **Soluzione:**

  Si creano interfacce per ogni tipo di prodotto.

  Ci saranno poi concreti prodotti che implementano queste interfacce.

  Queste stesse consentiranno ai clienti di fare uso dei prodotti.

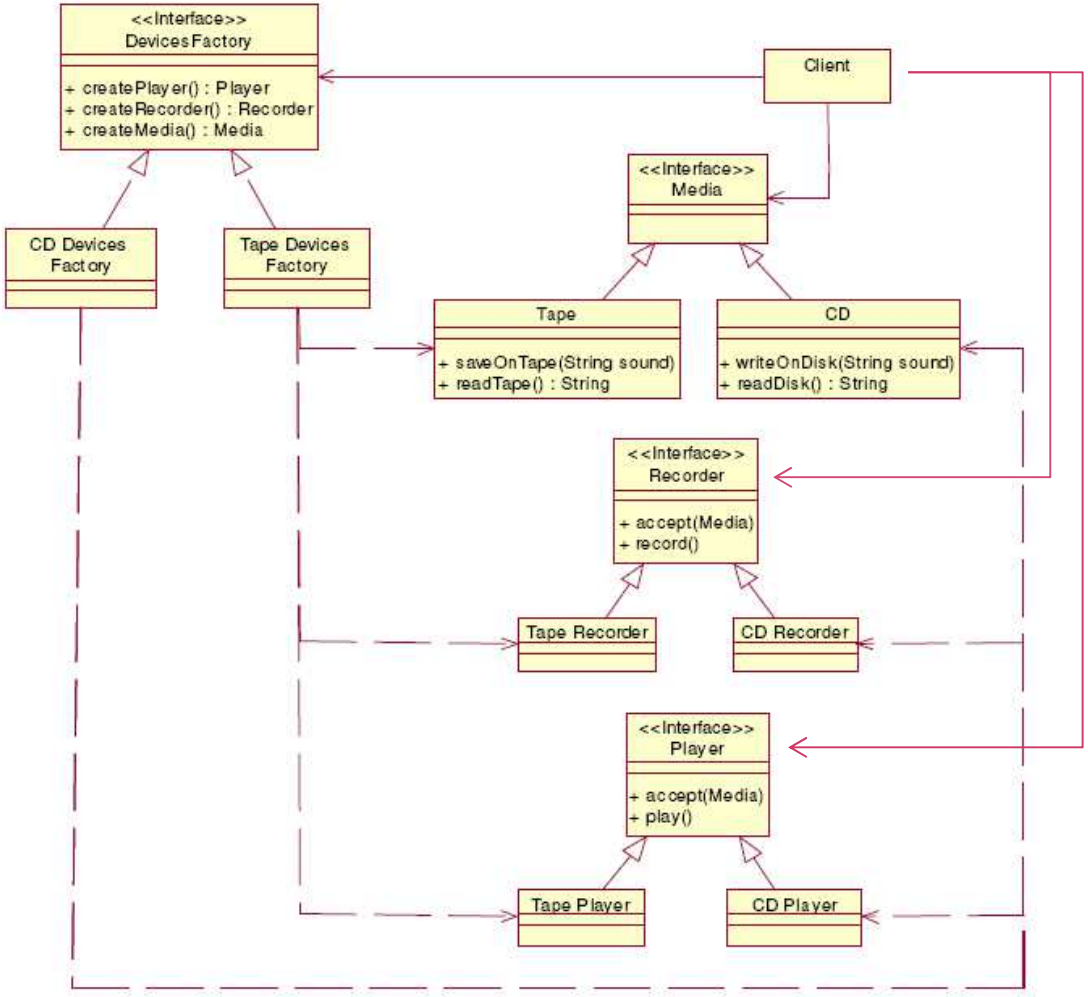
  Le famiglie di prodotti saranno create da un oggetto *factory*.

  Il client userà la factory opportuna per creare e disporre della famiglia di prodotti che vuole usare.

  Per generalizzare ulteriormente, le *factory* implementeranno una interfaccia comune di cui il Cliente sarà a conoscenza.

3) Abstract Factory

- Schema del modello
- Partecipanti:
  - **AbstractFactory**: interfaccia DevicesFactory
  - **ConcreteFactory**: classi TapeDevicesFactory e CDDevicesFactory
  - **AbstractProduct**: interfacce Media, Recorder e Player
  - **ConcreteProduct**: classi Tape, TapeRecorder, TapePlayer, CD, CDRecorder e CDPlayer
  - **Client**: classe Client.

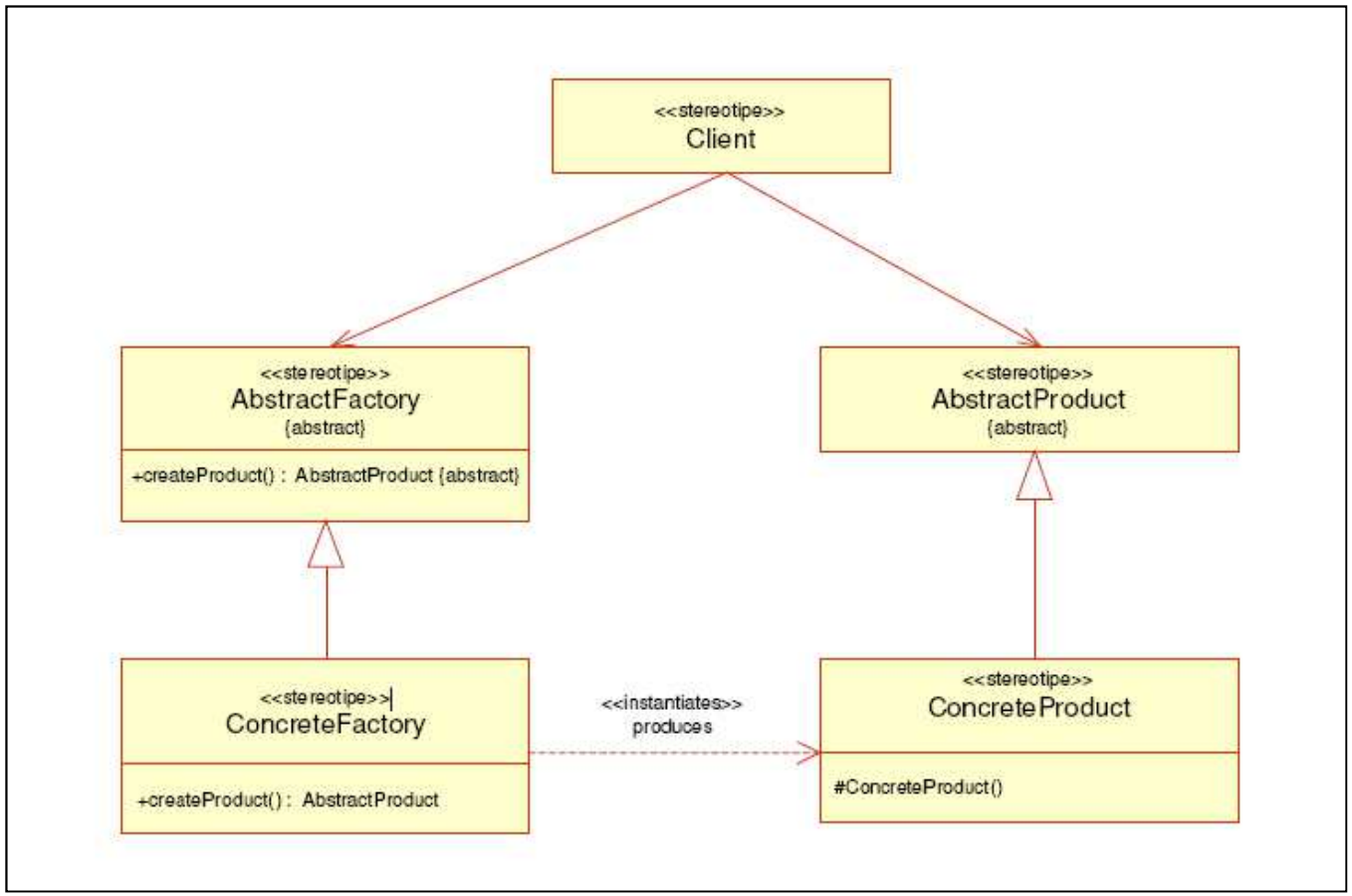


3) Abstract Factory

- **Problema:**
  - Si vuole creare una famiglia di oggetti (dipendenti tra loro) di cui staticamente si conosce l'interfaccia (o la super classe) ma non la classe effettiva.
  - Si vuole creare una libreria di classi di cui si espone solo l'interfaccia, ma non l'implementazione

- **Soluzione:**
  - Prevedere un'interfaccia per la creazione di famiglie complete di oggetti
  - Il client, che le utilizza, conosce solo come usarle

3) Abstract Factory Class Diagram



- Il **Client** crea una **ConcreteFactory**, ma poi usa i generici metodi per ottenere i singoli elementi della famiglia
- Gli oggetti concreti infine vengono usati come semplici **Product**

3) Abstract Factory

Conseguenze:

- Si vincola il client (consumatore dei prodotti) a creare (e usare) soltanto prodotti vincolati fra di loro.
- Lo stesso cliente può utilizzare diverse famiglie di prodotti
- Si può vedere come una generalizzazione del Factory Method su N classi Product

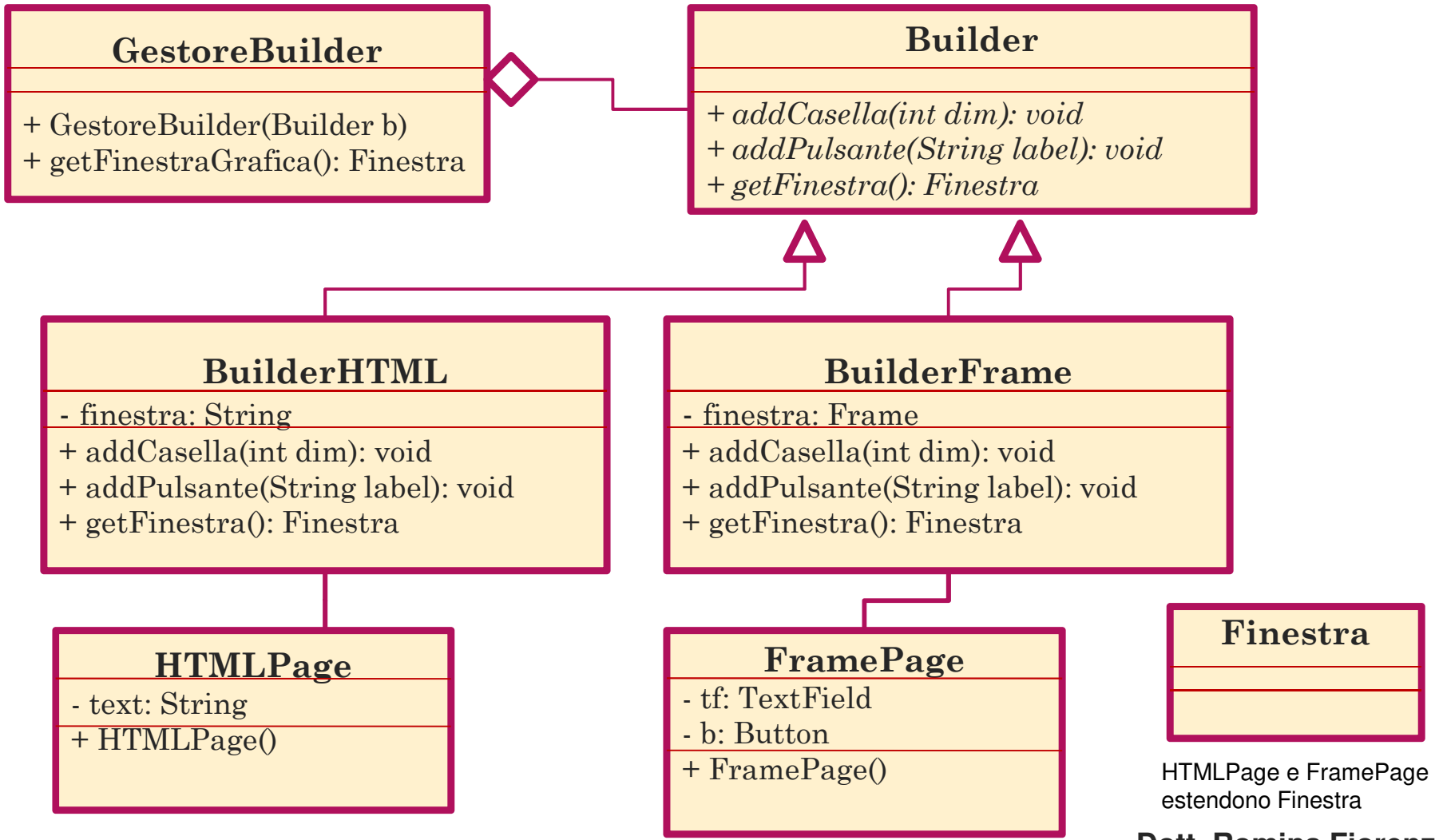
4) Builder

Esempio:

- Si vuole realizzare un'interfaccia grafica che presenti una casella di testo ed un pulsante.
- Si desidera però disporre di 2 versioni:
  - Una versione console → finestra grafica
  - Una versione web → form HTML

- L'oggetto di cui disporre è concettualmente uno solo, ma si presenta con rappresentazioni diverse.

4) Builder



HTMLPage e FramePage estendono Finestra

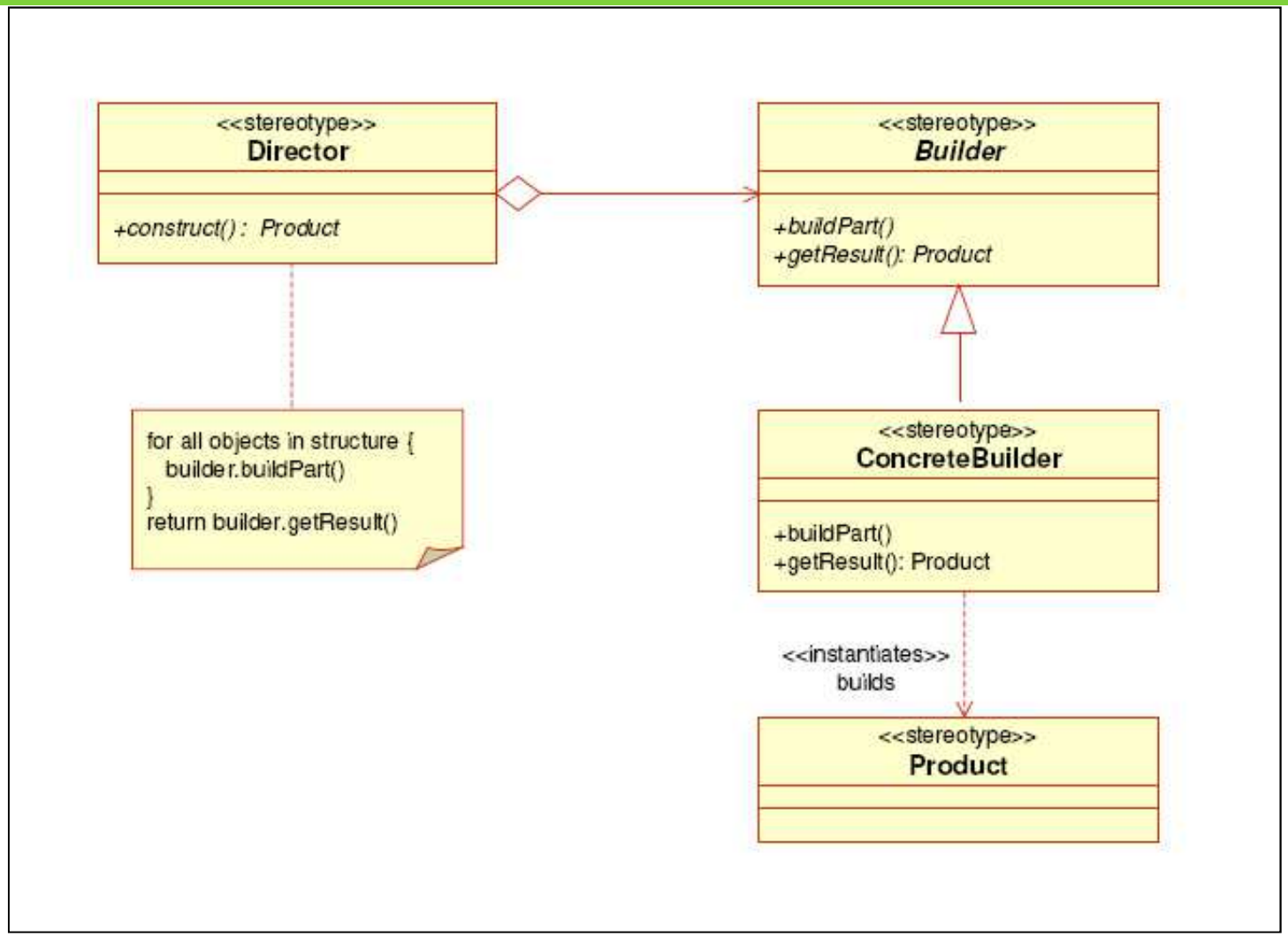
4) Builder

- Siamo costretti a creare 2 classi che rappresentino le 2 versioni della finestra grafica.
  - Queste classi avranno caratteristiche diverse, ma stessa modalità di creazione.
- Per questo, è necessario realizzare 2 classi Builder che si occupino di "fabbricare" le varie parti della finestra secondo le diverse caratteristiche.
- La logica di costruzione dell'oggetto viene così separata dalla sua rappresentazione.
- Inoltre l'algoritmo di costruzione viene centralizzato in una classe esterna, Director, che coordina le azioni da compiere.

4) Builder

- **Problema:**
  - Dobbiamo creare un oggetto con una struttura complessa
  - L'oggetto si può presentare con diverse rappresentazioni
  - Vogliamo separare l'algoritmo di costruzione della struttura dalla creazione dei singoli pezzi.
- **Soluzione:**
  - La responsabilità della creazione tra i vari oggetti , si concentra in un unico punto (classe Coordinatore) che conterrà la logica di creazione
  - Le classi delegate della reale costruzione saranno le classi Builder.

4) Builder Class Diagram



4) Builder

Conseguenze:

- Si rende la struttura più flessibile ad eventuali modifiche
  - E' possibile aggiungere nuove rappresentazioni degli oggetti, senza modificare la logica di costruzione
- Si può facilmente agire sul processo di creazione degli oggetti
  - Le singole parti si fabbricano separatamente
  - L'algoritmo è concentrato nel Director
- Si può utilizzare per creare oggetti senza avere in anticipo tutti i parametri per la costruzione
  - L'oggetto è completamente costruito solo alla fine della fabbricazione da parte dei Builder

5) Singleton

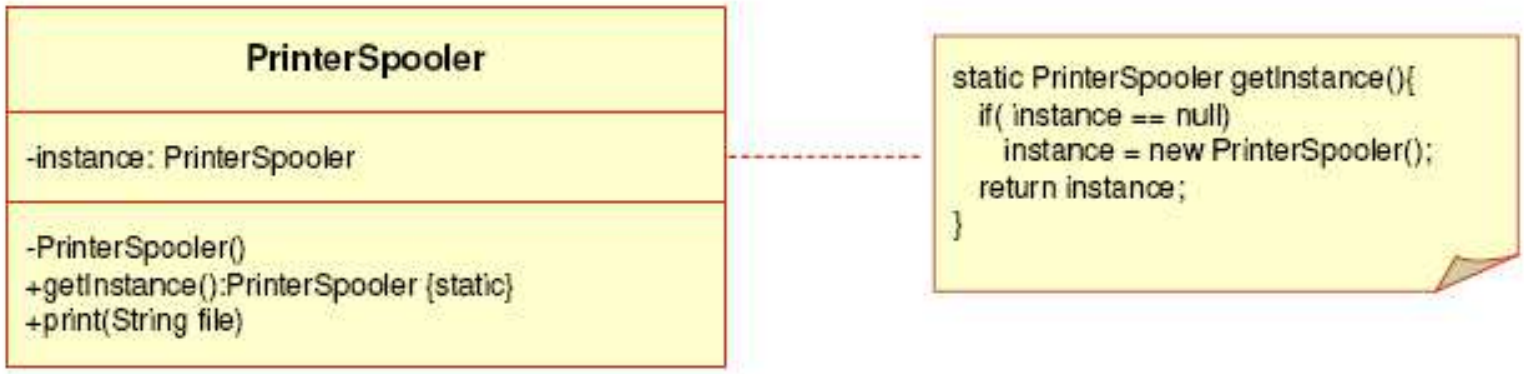
- **Esempio:**

  Un applicativo deve istanziare un oggetto che gestisce una stampante.

  Questo oggetto deve essere unico perché si dispone di una sola risorsa di stampa.

- La classe da creare deve garantire:
  - la creazione di un'unica istanza all'interno del programma
  - la condivisione da parte del programma della risorsa di stampa

5) Singleton


PrinterSpooler
-instance: PrinterSpooler
-PrinterSpooler()
+getInstance():PrinterSpooler {static}
+print(String file)
static PrinterSpooler getInstance(){
if(instance == null)
instance = new PrinterSpooler();
return instance;
}

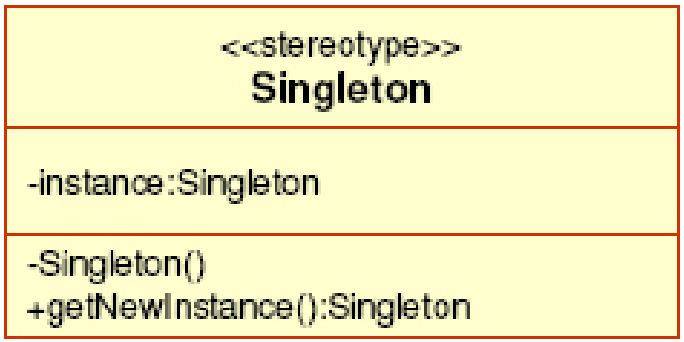
5) Singleton

- **Problema:**
  - Si usa per garantire che esista una ed una sola istanza di una certa classe

- **Soluzione:**
  - Prevenire la possibilità di creare oggetti di una certa classe tramite il costruttore.
  - L'idioma Java per implementare il pattern del Singleton prevede di dichiarare
    - Un attributo privato e statico del tipo della classe
    - Un costruttore privato
    - Un metodo pubblico e statico che restituisce l'istanza richiesta (agendo all'occorrenza sul costruttore o sull'attributo)

5) Singleton Class Diagram

- **Metodi:**
- Il *costruttore di Singleton* è privato, verrà invocato solo tramite il metodo *getNewInstance*
- Tale metodo è pubblico e si occupa di gestire l'unicità dell'oggetto
- La prima volta lo crea e imposta l'attributo
- Le successive, trova l'attributo già settato, allora lo restituisce, senza istanziare nuovamente!



Idioma in java

Nota:

- Per completare l'idioma la classe dovrebbe essere marcata come non estendibile (in java si usa **final**)
  - In realtà è una precisazione che potrebbe essere omessa poiché l'aver marcato il costruttore private già comporta che non posso implementare nuove classi figlie.

- Se poi la classe eredita da una gerarchia che implementa l'interfaccia `Clonable` → *ridefinire il metodo* `clone` *- marcandolo public* - e sollevare l'eccezione `CloneNotSupportedException`.
  - *Viceversa si potrebbero generare nuove istanze per clonazione*

Esempio


```
class SuperPrintSpooler implements Cloneable{
   int id = 10;
   public Object clone() throws CloneNotSupportedException  {
         SuperPrintSpooler ss = (SuperPrintSpooler) super.clone();
         ss.id = this.id;
         return ss;
   }
   public static void main(String[] args) {
         PrintSpooler ps = PrintSpooler.getInstance();
         PrintSpooler pss= null;
         try {
                  pss = (PrintSpooler) ps.clone();
         } catch (CloneNotSupportedException e) {
                  System.out.println("oggetto non clonabile");
         }
}
```

In questo modo otterrei un clone dell'unica istanza e avrei trovato un modo di alterare la classe Singleton

Soluzione


```
class PrintSpooler extends SuperPrintSpooler{

   // codice del Singleton

   public Object clone() throws CloneNotSupportedException  {
         throw new CloneNotSupportedException() ;
   }
   public static void main(String[] args) {
         PrintSpooler ps = PrintSpooler.getInstance();
         PrintSpooler pss= null;
         try {
                  pss = (PrintSpooler) ps.clone();
         } catch (CloneNotSupportedException e) {
                  System.out.println("oggetto non clonabile");
         }
}
```

In questo modo otterrei un eccezione e non sarà possibile alterare la finalità della classe Singleton