Object Pool → Prevede una creazione anticipata di un insieme di oggetti di una determinata classe (la loro gestione attraverso una "piscina" da cui vengono presi e restituiti gli oggetti quando necessario)
↳ Utile quando la creazione degli oggetti è costosa in termini di TEMPO - RISORSE.

Un'object pool prevede la creazione anticipata di un insieme di oggetti di una determinata classe, e la loro gestione attraverso un pool (o una "piscina") da cui vengono presi e restituiti gli oggetti quando necessario.
L'idea alla base di un Object Pool è quella di ridurre il costo di creazione e distruzione degli oggetti, migliorando le prestazioni del sistema. Piuttosto che creare e distruggere continuamente nuovi oggetti, il pool mantiene un insieme di oggetti già istanziati, in modo da poterli riutilizzare più volte.

Il client che necessita di un oggetto richiede il prelievo dall'Object Pool, che gli restituisce un oggetto disponibile. Quando il client ha finito di utilizzare l'oggetto, lo restituisce al pool per consentire ad altri client di utilizzarlo. L'Object Pool è particolarmente utile quando la creazione degli oggetti è costosa in termini di tempo e risorse (ad esempio, connessioni a database o socket di rete)

Factory method e Object pool

Il factory method può implementare un object pool . I client fanno richieste di oggetti, come di consuetudine. Quando l'istanza non è più in uso la si restituisce all'object pool per permettere ad altri client di renderla utilizzabile. Lo stato dell'istanza va ri-scritto (resettato) e l'object pull, dovendo essere un'istanza unica, va implementato con un Singleton.

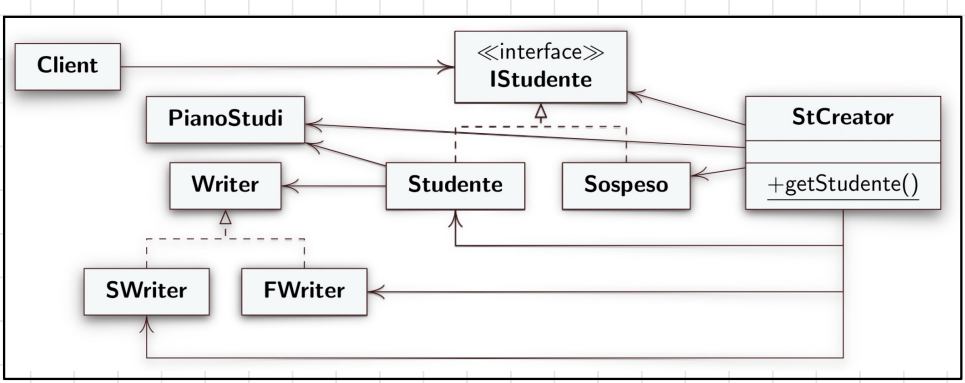
Un object pool può avere dimensione fissa oppure avere una dimensione variabile .

Dependancy Injection → Utilizzato per inviare le istanze create ad un'altra classe che le utilizzerà
Gli oggetti non vengono creati all'interno della classe questi oggetti vengono forniti da un componente esterno.

Il design pattern Factory Method può essere usato per inserire facilmente dipendenze necessarie alle istanze ConcreteProduct. Quando si utilizza il Factory Method insieme alla DI, si definisce un'interfaccia comune per l'oggetto da creare e si delega la scelta della classe concreta da utilizzare a un componente esterno, il cosiddetto "factory". In questo modo, il componente che utilizza l'oggetto non ha la responsabilità di conoscere le dipendenze dell'oggetto stesso e il processo di creazione viene separato dalla logica del programma.

Ad esempio, supponiamo di avere una classe Car che ha bisogno di un oggetto Engine per funzionare. Invece di creare l'oggetto Engine all'interno della classe Car, si può definire un'interfaccia comune EngineInterface e utilizzare un Factory Method per creare un'istanza di Engine quando necessario. Il Factory Method può essere implementato da una classe esterna chiamata CarFactory che conosce tutte le dipendenze necessarie per creare un'istanza di Car, inclusa l'istanza di Engine. In questo modo, la classe Car può essere costruita utilizzando l'istanza di Engine fornita dal Factory, senza preoccuparsi di come viene creato l'oggetto.

L'utilizzo della Dependency Injection insieme al Factory Method consente quindi di separare la logica di creazione degli oggetti dalla logica del programma, semplificando la manutenzione e il testing del codice.

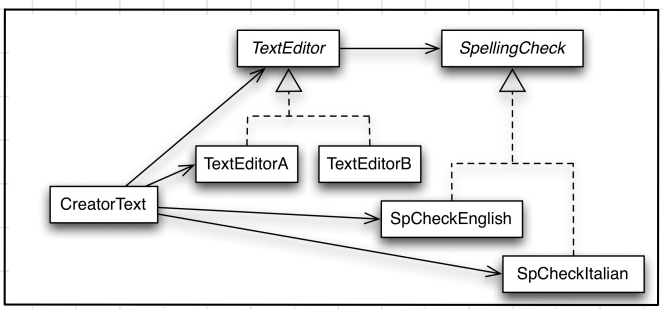


Una classe C usa un servizio S (ovvero C dipende da S). Esistono tante implementazioni di S (ovvero S1, S2), la classe C non deve dipendere dalle implementazioni S1, S2. Al momento di creare l'istanza di C, indico all'istanza di C con quale implementazione di S deve operare.

Esempio :

Una classe TextEditor usa un servizio SpellingCheck

- Ci sono tante classi che implementano il servizio SpellingCheck, in base alla lingua usata: SpCheckEnglish, SpCheckItalian, etc.
- TextEditor deve poter essere collegato ad una delle classi che implementano SpellingCheck




```
public class TextEditorA implements TextEditor {//TextEditorA è un Concrrroduct
   private SpellingCheck speller;
   public TextEditorA(SpellingCheck sp) {  // inserisco la dipen. fornendo sp,
       speller = sp;                       // istanza del serv., al costruttore
   }
   public void put(String s) {
       if (speller.check(s)) ...
       else ...
   }
}

public class CreatorText {  // CreatorText è un ConcreteCreator
   public static TextEditor getEnglishEditor() {
       return new TextEditorA(new SpCheckEnglish());
   }
   public static TextEditor getItalianEditor() {
       return new TextEditorB(new SpCheckItalian());
   }
}
```

Considerazioni

- L'attributo speller di TextEditorA è inizializzato attraverso l'invocazione del costruttore con l'opportuna istanza
- Il metodo getEnglishEditor() (metodo factory) conosce la classe per lo spelling da usare e inserisce (inject) la dipendenza sul concreteProduct TextEditorA
- Posso sostituire la classe che implementa SpellingCheck facilmente

Design pattern : Abstract Factory → Creazionale

Intento Consente di produrre famiglie di OGGETTI correlati senza specificare le loro classi CONCRETE

Abstract Factory è un modello di progettazione creazionale che consente di produrre famiglie di oggetti correlati senza specificare le loro classi concrete.

Problema → Abbiamo bisogno di un modo per CREARE singoli oggetti di arredo, in modo che corrispondano ad altri oggetti della stessa famiglia.

SEDIA VITTORIANA <-> SEDIA ANTICA (Sono sedie, ma di tipologia differente)

Immagina di creare un simulatore di negozio di mobili. Il tuo codice è composto da classi che rappresentano:

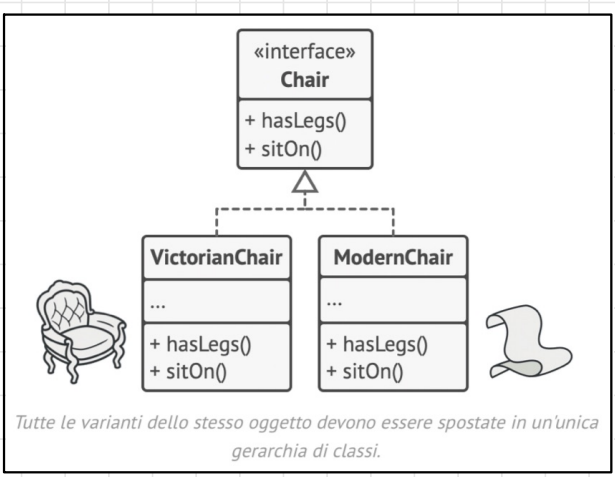
1 Una famiglia di prodotti correlati, ad esempio: Chair+ Sofa+ CoffeeTable.

2 Diverse varianti di questa famiglia. Ad esempio, i prodotti Chair+ Sofa+ CoffeeTable sono disponibili in queste varianti: Modern, Victorian, ArtDeco

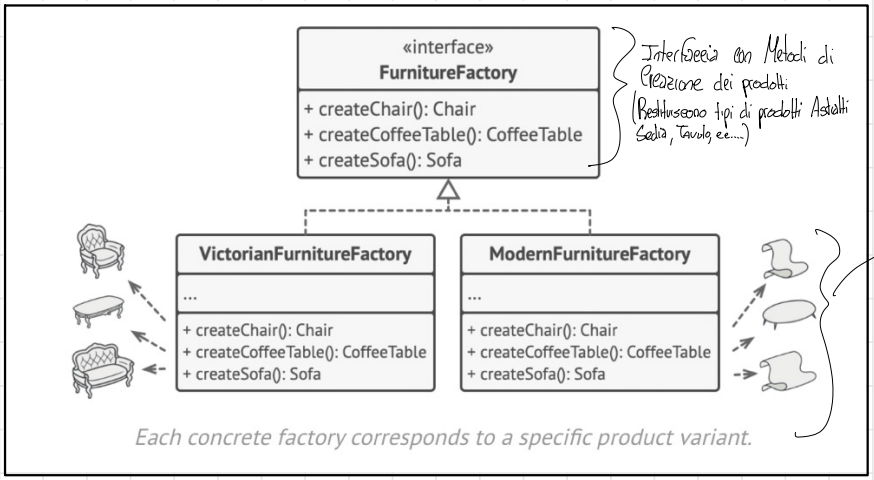
Hai bisogno di un modo per creare singoli oggetti di arredo in modo che corrispondano ad altri oggetti della stessa famiglia. I clienti si arrabbiano molto quando ricevono mobili non coordinati. Inoltre, non si desidera modificare il codice esistente quando si aggiungono nuovi prodotti o famiglie di prodotti al programma. I venditori di mobili aggiornano i loro cataloghi molto spesso e non vorrai cambiare il codice di base ogni volta che succede.

Soluzione Creiamo un interfaccia per ogni tipo di prodotto (Tavolo, Sedia, Divano); Fare in modo che tutte le varianti del prodotto seguano tali interfacce.

La prima cosa che suggerisce il modello Abstract Factory è dichiarare esplicitamente le interfacce per ogni prodotto distinto della famiglia di prodotti (ad esempio, sedia, divano o tavolino). Quindi puoi fare in modo che tutte le varianti dei prodotti seguano tali interfacce. Ad esempio, tutte le varianti di sedia possono implementare l' interfaccia Chair; Tutte le varianti di tavolino possono implementare l'interfaccia CoffeTable e così via.



La mossa successiva è dichiarare Abstract Factory , un'interfaccia con un elenco di metodi di creazione per tutti i prodotti che fanno parte della famiglia di prodotti (ad esempio createChair, createSofa e createCoffeeTable). Questi metodi devono restituire tipi di prodotto **astratti** rappresentati dalle interfacce estratte in precedenza: Chair, Sofa, CoffeeTablee così via.



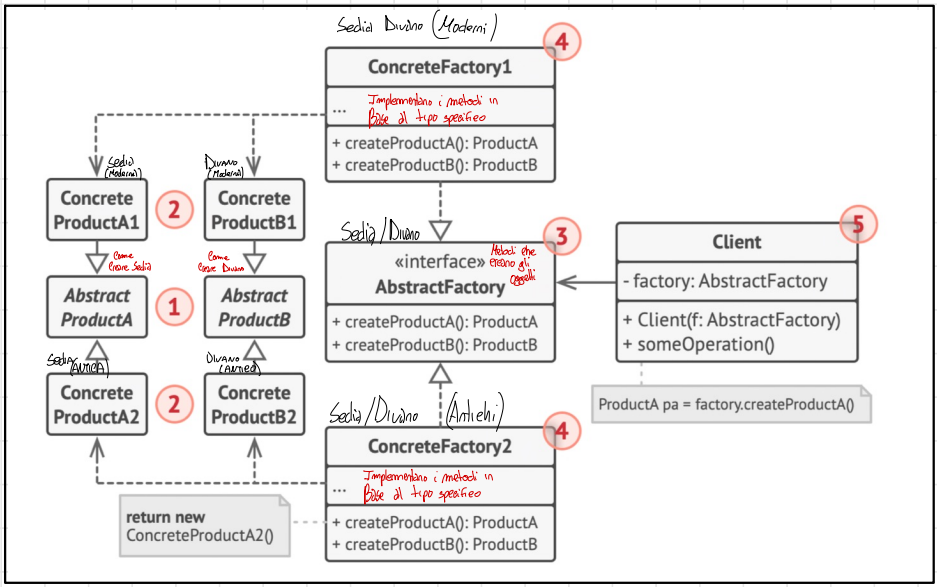
Ora, che ne dici delle varianti di prodotto? Per ogni variante di una famiglia di prodotti, creiamo una classe di fabbrica separata basata sull'interfaccia `AbstractFactory`. Una factory (fabbrica) è una classe che restituisce prodotti di un tipo particolare. Ad esempio, `ModernFurnitureFactory` possono solo creare oggetti `ModernChair`, `ModernSofa` e `ModernCoffeeTable`.

Il codice client deve funzionare sia con factory che con prodotti tramite le rispettive interfacce astratte. Ciò consente di modificare il tipo di fabbrica che si passa al codice cliente, nonché la variante di prodotto ricevuta dal codice cliente, senza violare il codice cliente vero e proprio.

Supponiamo che il cliente voglia una fabbrica per produrre una sedia. Il cliente non deve essere consapevole della classe della fabbrica, né importa che tipo di sedia ottiene. Che si tratti di un modello moderno o di una sedia in stile vittoriano, il cliente deve trattare tutte le sedie allo stesso modo, utilizzando l'interfaccia astratta `Chair`. Con questo approccio, l'unica cosa che il cliente sa della sedia è che in qualche modo implementa il metodo `siton`. Inoltre, qualunque variante della sedia venga restituita, si abbinerà sempre al tipo di divano o tavolino prodotto dallo stesso oggetto di fabbrica.

C'è un'altra cosa da chiarire: se il client è esposto solo alle interfacce astratte, cosa crea gli effettivi oggetti factory? Di solito, l'applicazione crea un oggetto factory concreto nella fase di inizializzazione. Poco prima, l'app deve selezionare il tipo di fabbrica a seconda della configurazione o delle impostazioni dell'ambiente.

Struttura



1) Gli **Abstract Products** dichiarano le interfacce per un insieme di prodotti distinti ma correlati che costituiscono una famiglia di prodotti.

2) I **Concrete Products** sono varie implementazioni di prodotti astratti, raggruppati per varianti. Ogni prodotto astratto (sedia/divano) deve essere implementato in tutte le varianti date (Vittoriano/Moderno).

3) L'interfaccia **Abstract Factory** dichiara un insieme di metodi per la creazione di ciascuno dei prodotti astratti.

4) Le **Concrete Factories** implementano i metodi di creazione della Abstract Factory . Ogni concrete factory corrisponde a una specifica variante di prodotti e crea solo quelle varianti di prodotto.

5) Sebbene le concrete factory creino istanze di prodotti concreti, le firme dei loro metodi di creazione devono restituire prodotti astratti corrispondenti. In questo modo il codice cliente che utilizza una fabbrica non viene accoppiato alla variante specifica del prodotto che riceve da una fabbrica. Il **Client** può lavorare con qualsiasi variante concreta di fabbrica/prodotto, purché comunichi con i propri oggetti tramite interfacce astratte.

Applicabilità Quando abbiamo una classe con un set di Factory Method che offusca la sua responsabilità PRIMARIA.

Usa Abstract Factory quando il tuo codice deve funzionare con varie famiglie di prodotti correlati, ma non vuoi che dipenda dalle classi concrete di quei prodotti: potrebbero essere sconosciuti in anticipo o semplicemente vuoi consentire l'estensibilità futura.

Prendi in considerazione l'implementazione di Abstract Factory quando hai una classe con un set di Factory Method che offusca la sua responsabilità primaria.

Prima vediamo che prodotti vi sono, poi definiamo le interfacce di prodotti astratti, InterGeea, Fabrica Astratta con i metodi Creazionali per tutti i prodotti. Implementiamo una Fabbrica per ogni tipo

Implementazione Passiamo l'oggetto Factory a tutte le classi che creano prodotti. Sostituiamo le chiamate dirette dei costruttori con chiamate al metodo appropriato (sull'oggetto Factory).

1. Mappa una matrice di tipi di prodotto distinti rispetto alle varianti di questi prodotti.
2. Dichiarare le interfacce di prodotto astratte per tutti i tipi di prodotto. Quindi fai in modo che tutte le classi di prodotti concreti implementino queste interfacce.
3. Dichiara l'interfaccia della fabbrica astratta con una serie di metodi di creazione per tutti i prodotti astratti.
4. Implementa una serie di classi di fabbrica, una per ogni variante di prodotto.
5. Crea il codice di inizializzazione di fabbrica da qualche parte nell'app. Dovrebbe creare un'istanza di una delle classi concrete factory, a seconda della configurazione dell'applicazione o dell'ambiente corrente. Passa questo oggetto factory a tutte le classi che costruiscono prodotti.
6. Scansiona il codice e trova tutte le chiamate dirette ai costruttori di prodotti. Sostituiscili con chiamate al metodo di creazione appropriato sull'oggetto factory.

Vantaggi Gli oggetti sono compatibili tra loro, Principio Responsabilità Unica, Open/Close,

Puoi essere certo che i prodotti che ricevi da una fabbrica sono compatibili tra loro.

Eviti lo stretto accoppiamento tra prodotti concreti e codice cliente.

Principio di responsabilità unica . Puoi estrarre il codice di creazione del prodotto in un'unica posizione, semplificando il supporto del codice.

Principio open/close . Puoi introdurre nuove varianti di prodotti senza violare il codice cliente esistente.

Svantaggi → Codice più Complesso a causa di sottoclassi, InterGeee.

Il codice potrebbe diventare più complicato di quanto dovrebbe essere, poiché insieme al modello vengono introdotte molte nuove interfacce e classi.

Differenza tra Abstract Factory e Factory Method

Sia il Factory Method che l'Abstract Factory sono design pattern creazionali utilizzati per creare oggetti in modo flessibile e senza accoppiamento rigido tra le classi coinvolte. Tuttavia, ci sono alcune differenze significative tra i due pattern.

Il Factory Method definisce un'interfaccia per creare un oggetto, ma delega la decisione sulla classe concreta da istanziare alle sottoclassi. In questo modo, la classe che utilizza il Factory Method non ha bisogno di conoscere la classe effettiva dell'oggetto creato, ma si affida al metodo creativo dell'oggetto.

L'Abstract Factory, d'altra parte, astrae la creazione di un insieme di oggetti correlati senza specificare le classi concrete. Definisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti tra loro, senza specificare classi concrete. Questo consente di creare interfacce di fabbrica multiple che possono creare diverse famiglie di oggetti. In altre parole, l'Abstract Factory fornisce un'interfaccia per creare gruppi di oggetti correlati, mentre il Factory Method si concentra sulla creazione di un singolo oggetto

28-03-2023

Hash Map

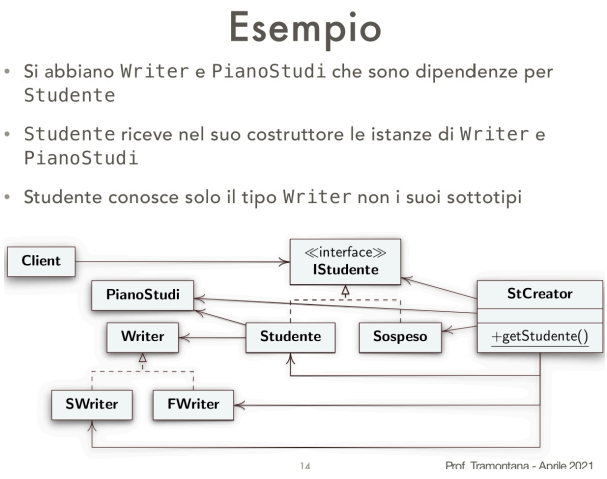
`HasMap` è una mappa associativa, formata da coppie di valori: **il primo valore** è la **chiave di accesso** per il secondo valore
Operazioni

- Inserimento in tabella: `put(chiave,valore)`
- Trovare il valore in base a una chiave: `valore= tabella.get(chiave)`
- Rimuovere un valore dalla tabella: `valore=tabella.remove(chiave)`

Aver separato Creator e ConcreteCreator introduce il meccanismo Dependency Injection: La classe MainEsami crea istanza di StCreator e la fa conoscere al Client. Il Client si lega e Creator e non a StCreator. In particolare:

```
public class MainEsami{
        public static void main(String[] args){
                Creator crea = new StCreator();
                Clienc c = new Client(crea);
        }
}
```

- Creo una sola istanza di StCreator e la passo al Client
- Si separa la decisone sulla costruzione dall'uso di quell'istanza
- Il client non deve sapere le istanze create ma deve conoscere solo l'interfaccia. (incapsulamento)
- Le dipendenze sono iniettate al client per mezzo di parametri nel suo costruttore. Questo permette di evitare complicazioni derivanti da metodi setter e da controlli per verificare che le dipendenze non siano null, di conseguenza il codice è più semplice
- L'oggetto che fa Dependency Injection si occupa di connettere (*fa **wiring** di*) varie istanze. In un unico posto vediamo le connessioni fra gli oggetti.



Abstract Factory

Intento

Fornire un'interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete

- Si hanno più gerarchie di Product e ConcreteProduct.
- Product è un'interfaccia totalmente diversa da factory method
- Quind ho **più interfacce** di Product:
  - un'interfaccia **ProductB** che implementa **ConcreteProductB1** e **ConcreteProductB2**
  - un'interfaccia **ProductA** che implemente **ConcreteProductA1** e **ConcreteProductA2**
- Quando seleziono ConcreteProductA2 è giusto **avere una corrispondenza** con ConcreteProductB2 (o anche altre relazioni)
  - Quando uso B2 mi relaziono solo a A2 o viceversa e non relazionarsi a A1 o B1

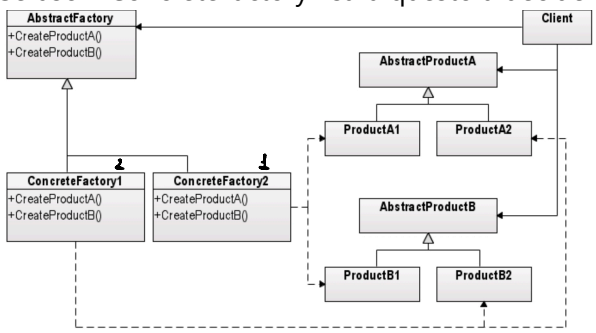
Problema

- Il sistema complessivo dovrebbe essere indipendente dalle classi usae, così da essere configurabile con una di varie famiglie di classi. Le classi di una famiglia dovrebbero essere usate insieme (in modo consistente)

> *Es. lo strato di interfaccia utente permette vari tipi di look-and-feel*

Soluzione

1. Creare delle gerarchie con interfacce AbstracProductB e AbstractProductA
2. Gestire la parte di creazione di istanze AbstractFactory e ConcreteFactory
3. Se uso il ConcreteFactory1 sarà questo a decidere la consistenza.



Usare una **serie di ruoli**: AbstractFactory, ConcreteFactory, AbstractProduct

- Si possono creare istanze di famiglie di classi in maniera consistente.
- Le famiglie di classi sono facilmente intercambiabili.
  - Se voglio eliminare una famiglia di AbstractProduct e cambiare (Per esempio un bottone cliccabile) in un bottone che scompare, allora posso inserire una nuova famiglia con un'interfaccia. In questo caso devo solo modificare il ConcreteFactory far istaziare le nuove classi create.

Maggiori informazioni su Wiki

> *Se volessi, quindi, AbstractProductC basta fare le operazioni sopra citate, quindi creare una nuova gerarchia.*

- **Uso di classi di famiglie diverse corrisponde a diverse famiglie di classi.**

Object Pool

Si ha bisogno di tenere le istanze (in una lista) create a run-time per riusare la stessa istanza in circostanze diverse. Questa tecnica è detta **Objcet Pool**

> *Quando si finisce di usare un'istanza, potrebbe venir restituita (venendo **liberata**) e **riutilizzata**.*

Utile per:

- Creazione di istanze può essere **pesante a livello computazionale**.
- Quando le istanze sono "libere" allora possono essere fornite a chi vuole una nuova istanza
- Si evita di creare altri spazi di memoria

Serve:

- Un posto per tenere le istanze (lista delle istanze create) messa in una classe che ha appunto tale ruolo (all'interno del factoryMethod).

```java
import java.util.LinkedList;
// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool extends ShapeCreator {
  private LinkedList<Shape> pool = new LinkedList<Shape>();
  // getShape() è un metodo factory che ritorna un oggetto preleavto dal pool
  public Shape getShape() {
      Shape s;
      if (pool.size() > 0) s = pool.remove();
      else s = new Circle();
      return s;
  }
  // releaseShape() inserisce un oggetto nel pool
  public void releaseShape(Shape s) {
      pool.add(s);
  }
}
```

E. Tramontana Pool, Dependency - 12 May 10 2

- Queste tecnica può **non avere limiti** oppure **può averne** (in base ai requisiti)
-