Esercitazioni di Ingegneria del Software

Design Patterns[1] in Java

Nico Pellegrinelli

Università degli Studi di Bergamo

1° semestre a.a. 2024/2025

[1] Materiale adattato da `https://refactoring.guru/design-patterns`

Section 1

Introduzione ai Design Pattern

Cosa sono (1)

- **Soluzioni** *tipiche* e *riutilizzabili* a **problemi** di design del software *noti* e *ricorrenti*.
- Non sono specifici pezzi di codice che si possono prendere e utilizzare *off-the-shelf* come le librerie, ma sono concetti generali da **applicare** e **adattare** al proprio codice.

Cosa sono (2)

- I design pattern si applicano a livello di **componente** (in Java quindi, a livello di package o gruppo di Classi). Infatti, spesso, sono descritti con semplici *Class Diagram*.
- Esistono anche gli **Architectural Pattern** (o styles), che si applicano a livello di sistema. Gli architectural pattern descrivono pattern comuni dell'organizzazione della struttura del sistema (e.g. layers, microservizi, MVC) $=>$ maggior livello di astrazione.

Cosa sono (3)

Struttura generale per descrivere un Design Pattern:

- **Context**
- **Problem**
- **Forces** (fattori e vincoli che influenzano la scelta)
- **Solution** (spesso un Class Diagram)

Opzionali:

- **Anti-patterns**
- **Related patterns**
- **References**

Storia

- L'idea di pattern è stata introdotta da Christopher Alexander nel contesto dell'**architettura** (pattern per definire l'altezza delle finestre, la dimensione delle aree verdi, etc.) nel 1977.
- Nell'ambito della programmazione, l'idea viene ripresa per **OOP** da Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm nel 1994 con il libro *Design Patterns: Elements of Reusable Object-Oriented Software* (detto anche "*The Book by the Gang of Four*").

Perché utilizzarli

- Sono soluzioni provate e **testate** a problemi comuni. Di fronte ad un problema, il programmatore che conosce i Design Pattern ha più **strumenti** nella sua mano per risolverlo.
- Facilitano la **comunicazione** nel team.
- Studiare i pattern permette di **imparare** dall'esperienza maturata negli anni da esperti di programmazione! Insegnano come risolvere i problemi utilizzando i principi dell'**object-oriented** design.

Critiche

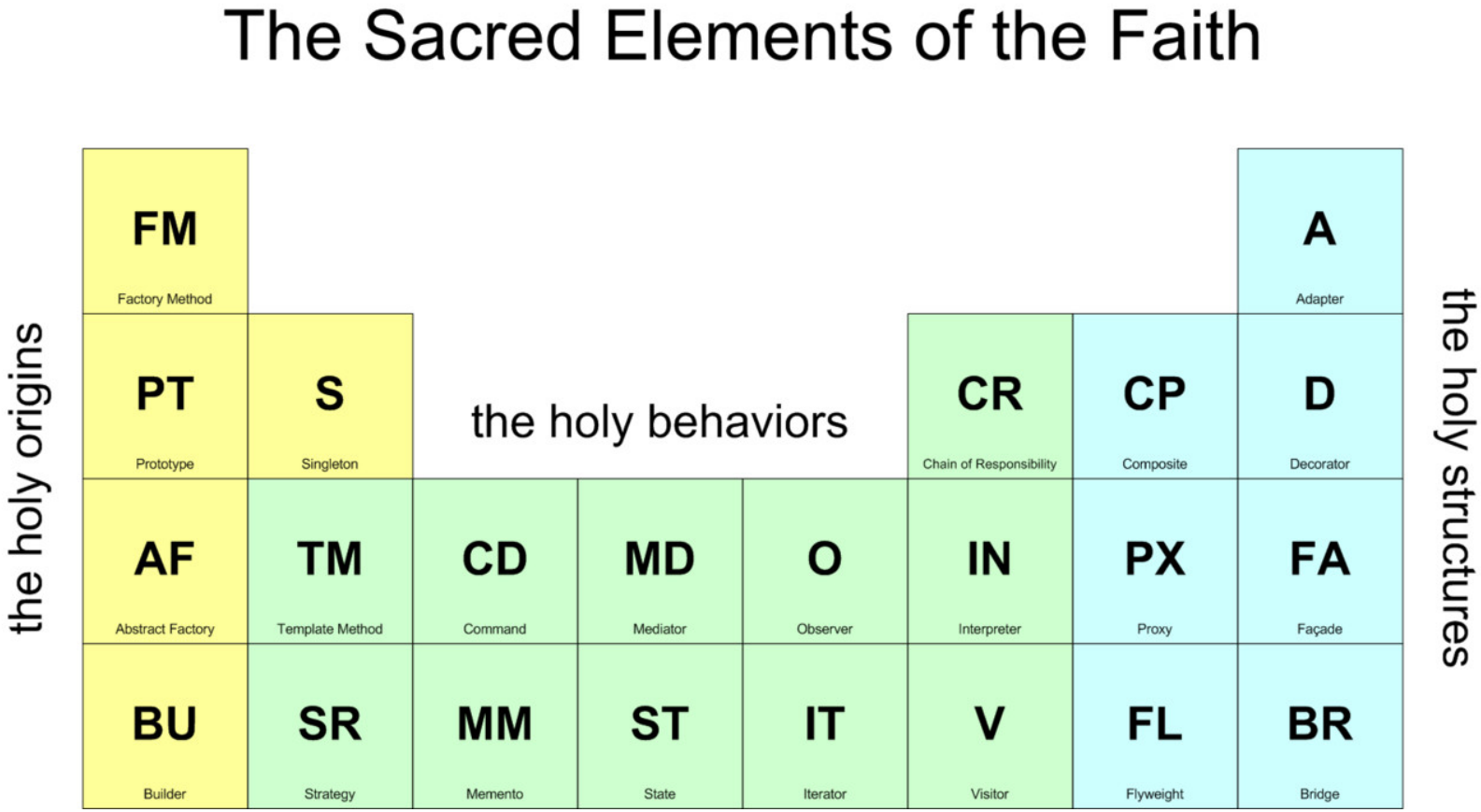
- Se applicati alla lettera, senza **adattarli** al contesto, possono risultare inefficienti.
- Utilizzo non giustificato: rischio di applicare i pattern ovunque, anche dove del codice **più semplice** sarebbe meglio.
- La necessità di applicare un pattern nasconde l'utilizzo di un linguaggio di programmazione non potente abbastanza.

I primi due punti indicano, quindi, che l'applicazione dei design pattern deve essere **ragionata**!

Classificazione (1)

- **Creational**: meccanismi per la *creazione* di oggetti con l'obiettivi di incrementare flessibilità e riutilizzo di codice (e.g. Singleton, Factory).
- **Structural**: per *assemblare* oggetti e classi in strutture più grandi ma sempre flessibili ed efficienti (e.g. Adapter, Facade).
- **Behavioral**: meccanismi di *comunicazione* e di *assegnamento di responsabilità* tra oggetti (e.g. Observer, Strategy, Visitor).

Classificazione (2)



Section 2

Singleton

Singleton (1)

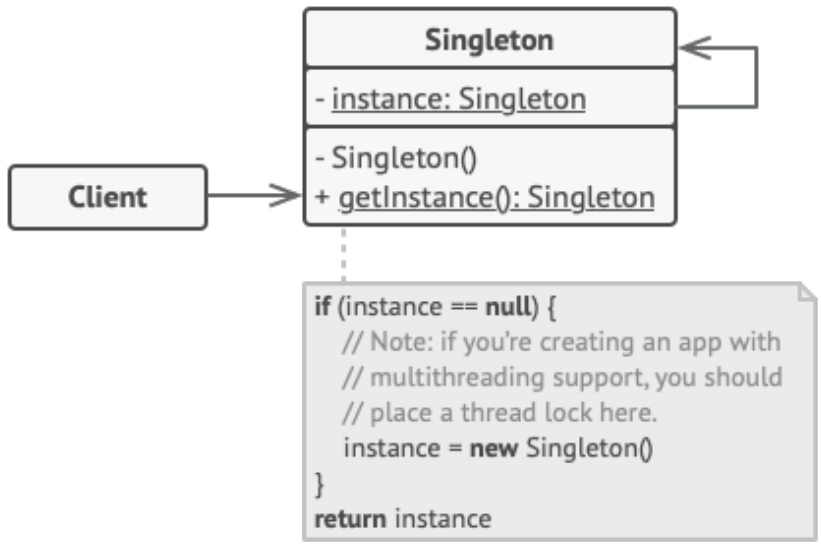
Creational

- **Contesto**: È molto comune avere classi per cui si vuole che possa esistere una sola istanza.
- **Problema**: Come assicurarsi che non esista mai più di un'istanza?
- **Forze**: La possibilità di accedere all'istanza deve essere possibile a tutte le classi che ne hanno bisogno (visibilità pubblica), ma se si usa un costruttore pubblico non c'è garanzia sul numero di istanze che si possono creare.

Singleton (2)

Creational

Il singleton permette di assicurarsi che una classe abbia al più **una sola istanza**.



Esercizio

Un'azienda vuole implementare un sistema di configurazione centralizzata per il proprio software, che deve mantenere un unico set di configurazioni condiviso da tutto il sistema. Tuttavia, se venissero create più istanze separate della classe di configurazione, potrebbero verificarsi inconsistenze nei dati.
Si crei una classe `ConfigurationManager` che deve:

- Contenere un insieme di impostazioni rappresentate da una `Map<String, String>`
- Offrire metodi per aggiungere, aggiornare e leggere le configurazioni

Si scriva un `main` in cui viene dimostrato il funzionamento del singleton.

Section 3

Adapter

Adapter (1)

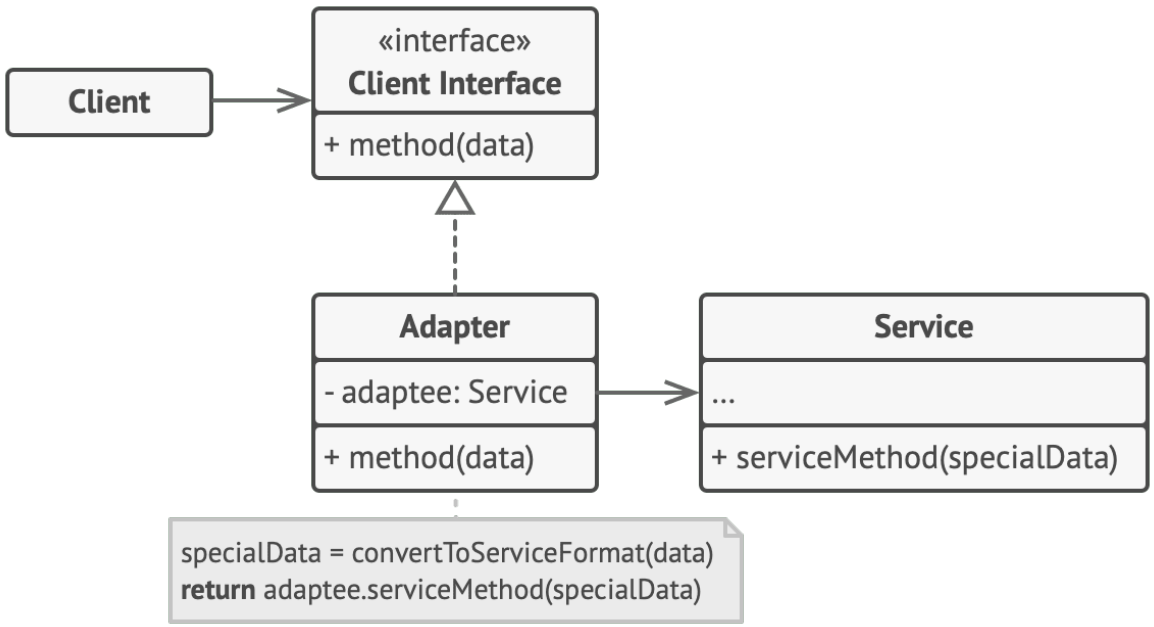
Structural

- **Contesto**: Stai costruendo una gerarchia di ereditarietà e vuoi integrare una classe esistente (magari scritta da qualcun altro) con signature dei metodi differenti. Spesso, la classe da riutilizzare è parte di una sua gerarchia.
- **Problema**: Come sfruttare il polimorfismo per riutilizzare questa classe anche se ha signature incompatibili?
- **Forze**: Non vuoi o non puoi utilizzare multiple inheritance (ad esempio perché stai programmando in Java).

Adapter (2)

Structural

L'Adapter risolve il problema dell'**incompatibilità tra interfacce diverse**.



Esercizio

Devi sviluppare un'applicazione che gestisce pagamenti. Hai una vecchia libreria che rappresenta un sistema di pagamento che accetta solo importi in centesimi (integer) e restituisce una stringa con la conferma del pagamento. D'altra parte, l'interfaccia moderna dell'applicazione lavora con importi in euro (double).
In particolare:

- Si implementi una classe `OldPaymentProcessor` contenente il metodo `String processPayment(int amountInCents)`
- Si implementi l'interfaccia `ModernPaymentProcessor` che espone il metodo `void processEuroPayment(double amountInEuros)`
- Si crei una classe `PaymentAdapter` che permetta di utilizzare `OldPaymentProcessor` come un `ModernPaymentProcessor`

Si scriva un `main` in cui viene dimostrato il funzionamento dell'adapter.

Section 4

Observer

Observer (1)

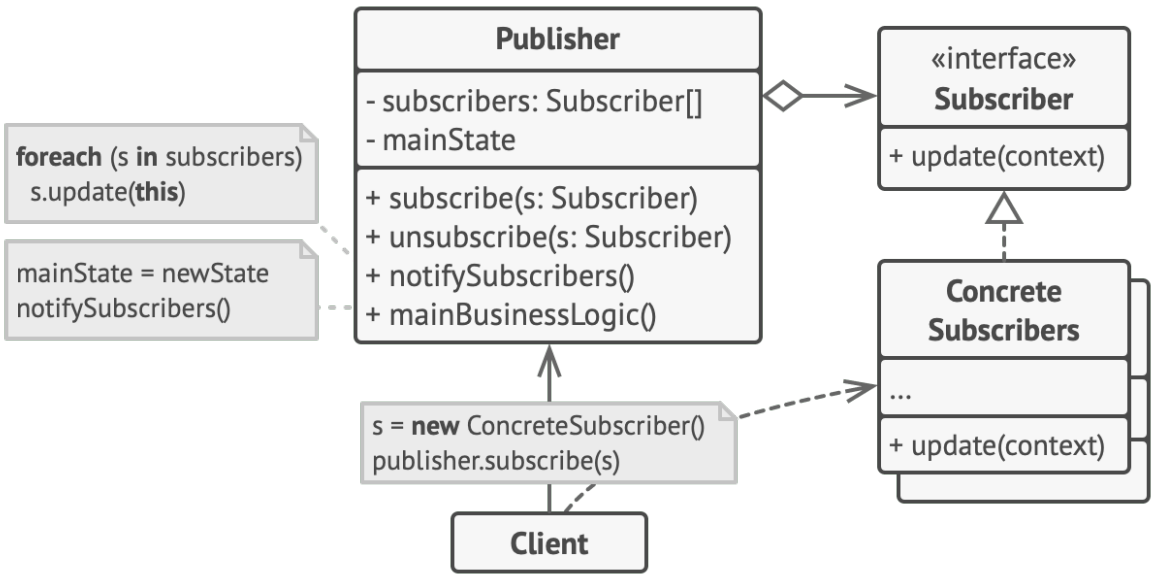
Behavioral

- **Contesto**: Quando si crea una two-way association tra due classi, il codice tra queste diventa inseparabile. Spesso, se ne modifico una, devo modficare l'altra di conseguenza.
- **Problema**: Come ridurre l'interconnessione tra le classi? Ossia, come permettere ad un oggetto di comunicare con altri oggetti senza che sappia di che classi sono?
- **Forze**: Si vuole massimizzare la flessibilità del sistema.

Observer (2)

Behavioral

L'Observer permette la comunicazione tra classi **riducendone l'interconnessione e il livello di accoppiamento**.



Publisher = Observable
Subscriber = Observer

Esercizio

Stai sviluppando un sistema per notificare i clienti di una banca quando il saldo del loro conto cambia. Il sistema deve supportare più osservatori (ad esempio, notifiche via SMS, notifiche via email). Ogni osservatore deve essere notificato automaticamente ogni volta che il saldo cambia.
In particolare:

- Si implementi una classe `BankAccount` (il Publisher o Observer), con un campo privato `balance`
- Si implementi l'interfaccia `Observer` (o Subscriber)
- Si creino due osservatori conreti: `EmailNotificator` e `SmsNotificator` che simulano l'invio di email e sms quando il `balance` di `BankAccount` cambia

Si scriva un `main` in cui viene dimostrato il funzionamento dell'observer.

Section 5

Strategy

Strategy (1)

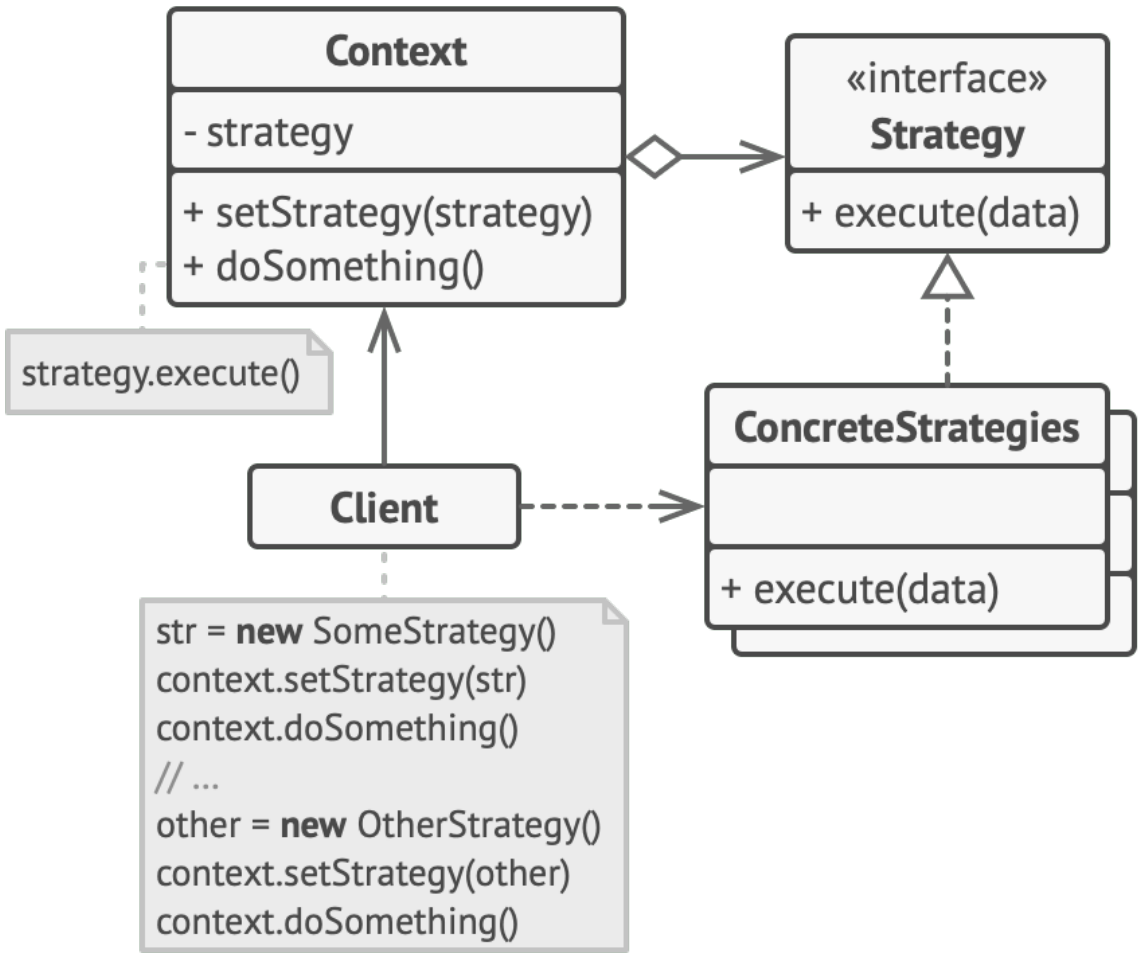
Behavioral

- **Contesto**: Si vuole realizzare un oggetto il cui comportamento/algoritmo possa cambiare dinamicamente (a runtime).
- **Problema**: Come assicurarsi che non ci siano impatti sul resto del sistema quando il numero di algoritmi cambia o l'implementazione di un algoritmo cambia?

Strategy (2)

Behavioral

Lo Strategy pattern permette di selezionare dinamicamente l'**algoritmo da utilizzare**.



Esercizio

Stai sviluppando un'applicazione per calcolare il prezzo finale di un prodotto con diversi tipi di sconti. A seconda della promozione in corso, il sistema deve applicare uno tra i seguenti algoritmi di sconto: sconto del 10%, sconto di 5€, nessuno sconto.
In particolare:

- Si implementi l'interfaccia `DiscountStrategy` che espone il metodo `double applyDiscount(double price)`
- Si creino tre strategie conrete: `PercentageDiscount`, `FixedAmountDiscount` e `NoDiscount`
- Si implementi una classe `PriceCalculator` che utilizzi una strategia di sconto per calcolare il prezzo finale

Si scriva un `main` in cui viene dimostrato il funzionamento dello strategy pattern.

Section 6

Factory

Factory (1)

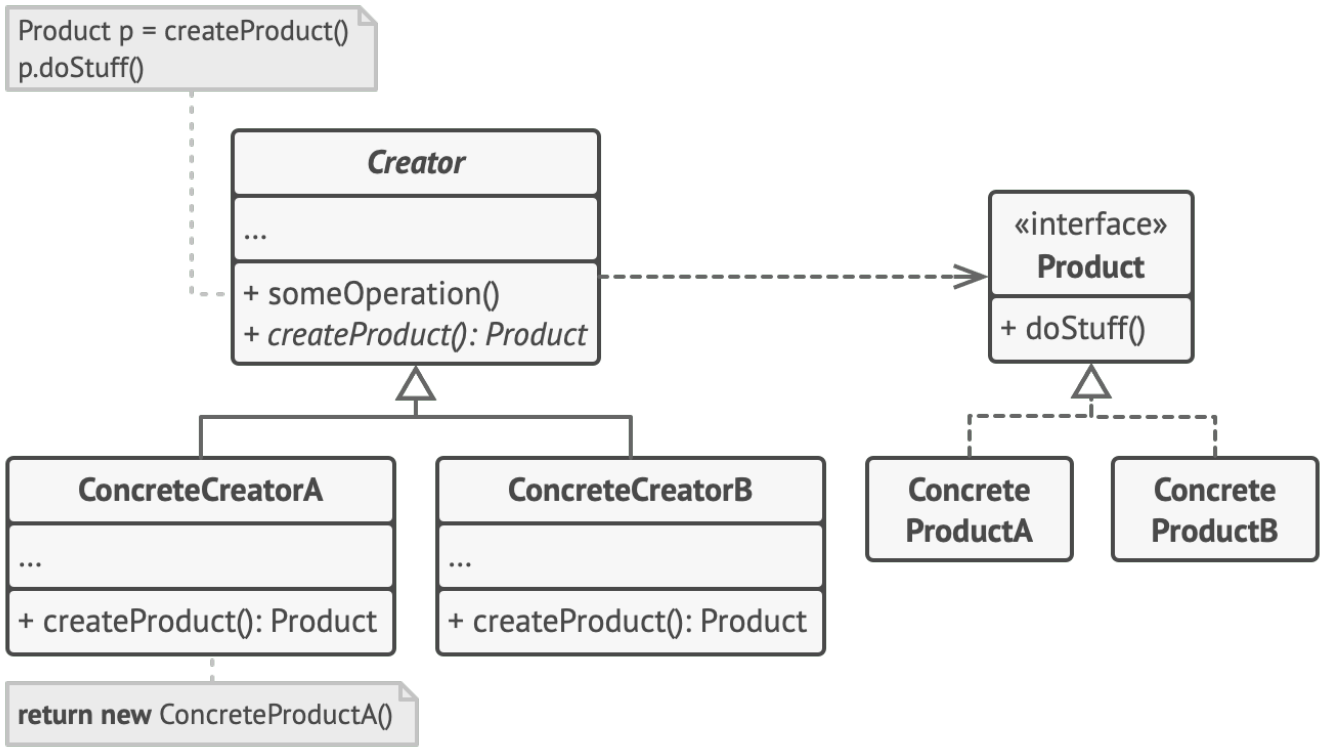
Creational

- **Contesto**: Hai un framework riutilizzabile che deve anche creare oggetti. Però, la classe degli oggetti da creare dipende dalla specifica applicazione.
- **Problema**: Come permettere ai programamtori di aggiungere una application-specific class in un sistema basato su questo framework?
- **Forze**: Si vogliono mantenere i benefici di usare un frameork riutilizzabile ma senza perdere la possibilità di lavorare con classi che il framework ancora non conosce.

Factory (2)

Creational

La Factory viene usato per **delegare la creazione di oggetti** a una classe specializzata, rendendo il sistema estensibile senza dipendere dalle implementazioni concrete.



Creator = Factory

Esercizio

Stai sviluppando un'applicazione per un sistema di gestione dei documenti. I documenti possono essere di diversi tipi: Word Document, PDF Document. Il sistema deve essere in grado di creare diversi tipi di documenti in base alle esigenze dell'utente senza che il codice principale conosca i dettagli della loro implementazione.
In particolare:

- Si implementi una classe astratta `Document` (con i metodi `open()` e `close()`) e le due classi concrete `WordDocument` e `PdfDocument`
- Si implementi la classe astratta `DocumentFactory` e le due implementazioni `WordDocumentFactory` e `PdfDocumentFactory`

Si scriva un `main` in cui viene dimostrato il funzionamento della factory.

Section 7

Esercizio Finale

Esercizio (1)

Stiamo sviluppando un piccolo gioco da tavolo digitale nel quale:

- Esiste un unico motore centrale di gioco con una lista di giocatori e una configurazione di gioco.
- I giocatori possono essere maghi o guerrieri.

Esercizio (2)

Si modifichi il codice presente nel progetto `BoardGame` nella repository:
`https://github.com/nicopellegrinelli/EsercitazioniIdS-public`
Applicando i Design Pattern (non per forza quelli visti sopra) che si ritengono adeguati, tenendo conto che:

- Deve essere possibile, per un giocatore, cambiare tra mago e guerriero a runtime.
- La configurazione di gioco deve essere accessibile ma non modificabile da altri componenti.

Soluzione di questo e degli altri esercizi nel branch `solution`:
`https://github.com/nicopellegrinelli/EsercitazioniIdS-public/tree/solution`